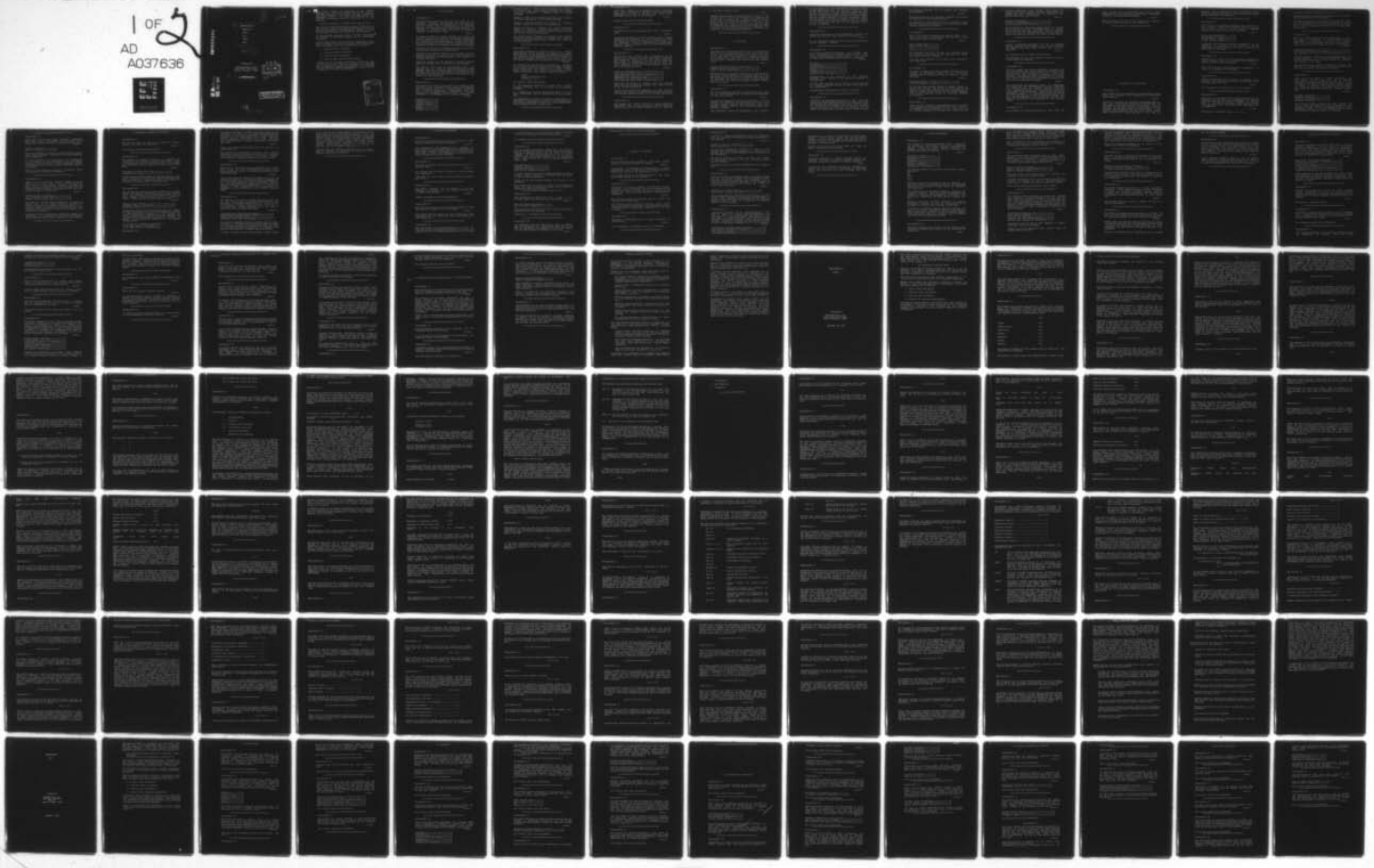
AD-A037 636 RL6 ASSOCIATES INC RESTON VA F/6 9/2
EVALUATION OF CORAL 66, PASCAL, CS-4, TACPOL, CMS-2.(U)
NOV 76

UNCLASSIFIED NL



(6)

EVALUATION

CORAL 66,

PASCAL,

CS-4,

TACPOL,

CMS-2.

(12) 18

Prepared b

RLG Associates,
11250 Roger Bacon
Reston, Virginia

(11) 18 November 18,

SIL

39

This report presents an evaluation of the CORAL66 language with the requirements listed in the "TINMAN". (DOD Requirements For High Order Computer Programming Languages, "TINMAN" - 1 March, 1976, Section IV). For purposes of comparison CORAL66 is considered to be defined by:

Official Definition of CORAL66 Third Edition, 1974

There are 78 language requirements listed in Section 4 of the "TINMAN". This report compares CORAL66 with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The introductory paragraph of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meanings are as follows:

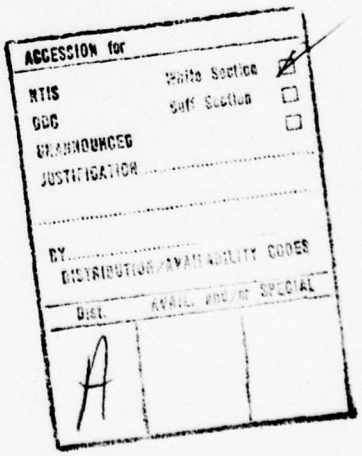
T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DOD "TINMAN" requirements.



A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

---p---

In general, CORAL66 is a typed language. Most types are known at compile time. However, two specific deficiencies exist in this area. Pointers are not defined explicitly as pointers but rather as integers. Floating point variables are declared as integer and use a software language extension to perform the necessary arithmetic operations.

Additionally, the OVERLAY construct provides a method for violating type checking and verification.

Addition of the keyword POINTER to the language, with appropriate changes to the syntax, will provide a method of uniquely identifying pointer variables. For detailed information see (D6).

Similarly, extension of the language to require specific typing of floating point numbers can be accomplished.

The OVERLAY option can be restructured within the language but the impact on existing programs or future programs operating on limited machine configurations may preclude this option. Development of a method to permit an overlay operation and also preserve type checking appears to be very expensive.

Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), boolean and character and will provide arrays (i.e., composite data structures with indexible components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

---p---

Integer.....T
Fixed point.....T
Floating point.....F
Character string.....T
Boolean.....F
Arrays.....P
Records.....P

Floating point -- loating point variables are declared as integer type in the language. Floating point operations are implemented as procedure calls to library routines.

Boolean -- There are no boolean variables and no bracketed boolean expressions in the CORAL66 language.

Arrays -- Arrays are provided for within the language. However, arrays are limited to two dimensions.

Records -- There is a provision for record structures within the language. Records are definable by a table mechanism. However, records do not define types, nor may they be used as type constructors.

The language can be modified to include both floating point and boolean capabilities at modest cost. Expansion of array and record capabilities, however will prove to be more expensive.

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---F---

The language does not include this capability. Inclusion of an optional single floating point precision declaration within a block that pertains to all local floating point identifiers will require an additional language keyword, PRECISION, which would be used as follows:

```
BEGIN
FLOATING PRECISION 1E-4 ;
FLOATING A, B, C ;
```

The following rules would apply:

1) All arithmetic operations on local (to a scope) floating variables will be truncated to the specified precision.

2) Operations on floating data between local and nonlocal identifiers will be truncated to the lower precision.

The requirement to enable individual specification of identifiers would still not be met. Precision specification would be considered to pertain to all floating identifiers declared within a given block.

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

---T---

CORAL66 totally meets this requirement.

Requirement: A5

Character sets will be treated as any other enumeration type.

---F---

A printing character is assumed to have a unique integer representation dependent on some hardware or software convention. The form is included within the integer syntax definition. Printing characters are implementation dependent. Layout characters may not be included as arguments of literals.

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

---P---

Number of dimensions fixed at compile time.....T
Type fixed at compile time.....I
Lower subscript bound fixed at compile time....I
Subscripts from contiguous range.....T
Subscripts from enumeration type.....P
Upper subscript bound fixed at scope entry.....F

Subscripts are confined to integer type. The language does not allow an explicit character as a subscript. See A5 for further information.

Arrays are limited to two dimensions and are fixed at compile time. Addition of variable upper bounds to arrays will require extensive change to the language definition.

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified

by the user at compile time.

---p---

Alternative structures are permitted by the use of the overlay feature and by use of pointers. Components must be specified at compile time. However, type checking and array bound protection are not secure. There are no facilities included within the language to identify type disagreements or array overruns or underruns. Inclusion of a discrimination feature will require extensive restructuring of the formal definition of the language, and indeed may not be economically feasible.

B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

---p---

Variable declaration for all data types.....T
Encapsulated type declaration.....F
Array or record declaration.....P

Assignment is by individual variable or by individual entry within tables (records). The language does not permit assignment of complete arrays or records. These aims may be approached by definition of user written procedures but still will not provide a consistent language definition and notation.

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---p---

The source language can be used to compare any two data objects for identity. The comparator works on a bit by bit basis without regard to type. This means that, conceivably, reals, integers and characters could be considered equivalent.

Type comparison, within the constraints of the "TINMAN",

for equivalence add a new dimension of difficulty. Limited type comparisons, when the absolute value of subscripts are known or when a specific equivalence relationship is explicitly stated, can be achieved. However, use of pointers and overlay features for element access provide an escape mechanism from type equivalence checking that cannot be completely avoided. Practically speaking, either the rules for type checking must be relaxed or the constructs permitted in the language must be severely restricted.

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---p---

All six relational operators are included. Unordered sets are not included in CORAL66.

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

---p---

Addition.....T
Subtraction.....T
Multiplication.....T
Division (with real result).....T
Negation.....T
Exponentiation.....F
Division with remainder.....F

Exponentiation is not supported in the language. Remainder after division is not explicitly available in the language.

Division with remainder can be included by the addition of a special remainder operator and the appropriate syntax modification.

Exponentiation capabilities can also be included.

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond

the specified precision will be allowed for floating point numbers.

---U---

Truncation rules are not explicitly discussed in the formal definition of the CORAL66 language.

The assumption is made that most of the truncation rules specified by this requirement are met since they are good compiler development practices.

Requirement: B6

The built-in boolean operators will include "AND", "OR", "NOT", and "NOR". The operations "AND" and "OR" will be evaluated in short circuit mode.

---P---

Short circuit "AND".....T
Short circuit "OR".....T
NOT.....F
NOR.....F

The operations "NOT" and "NOR" are not included within the language. Short circuit mode evaluation for scalars is defined within the language.

"NOT" and "NOR" operators can be added to the language at very modest cost.

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---F---

Assignment between records and arrays.....F
Scalar operations on arrays.....F

The language only supports element by element access of arrays and records. The capability can be added to the language. Again, type checking for some operations can be easily included but use of pointer and overlay capabilities can still provide a method to circumvent complete type checking.

Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit

conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

---P---

Explicit conversions.....T
No implicit conversions.....F
Explicit between scale factors.....T

The programmer can impose any desired system of evaluation by the use of Number type (Expression). If no explicit conversion rules are specified, implicit language rules are used for conversion. Changes to eliminate the implicit conversion capability will have minimal impact.

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

---F---

The language does not support this function.

The capability for run time exception checking can be included within the language.

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---F---

The language does not explicitly define I/O operations. All operations are accomplished by calls to procedures which are tailored to the required hardware configuration. Language constructs can be defined to satisfy this requirement. There are however, complications and machine dependency problems. Bit manipulation constructs provided by the language can be used to generate programs that are machine dependent. While this violates "TINMAN" constraints, this capability is necessary in many applications.

Requirement: B11

The language will provide operations on data types de-

fined as power sets of enumeration types (see B6). These operations will include union, intersection, difference, complement, and an element predicate.

---F---

Power sets are not defined in the language. This capability can be added with not too much difficulty.

C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

---P---

Although an algorithm for evaluating expressions does not form part of the formal definition of the language, all syntactically outermost terms in an expression will be evaluated to the required numeric type before the adding operators are applied. However, if an expression is enclosed in round brackets, its terms are no longer 'outermost' and the rule no longer applies. In this case the

algorithm for the particular compiler determines the sequence of events. Additionally, the programmer may impose any desired system of evaluation.

Requirement: C2

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---P---

Unambiguous presentation.....P
Few precedence levels.....T
Require explicit parentheses.....F
No user defined levels.....T

Precedence levels are those normally encountered in all languages. The language can be modified to require parentheses in certain cases if the list of cases can be agreed upon by all users.

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

---T---

There are no special case constraints on expressions included in the syntax of the language.

Requirement: C4

Constant expressions will be allowed in programs where constants are allowed, and constant expressions will be evaluated before run time.

---T---

The requirement is completely met by the language.

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

---P---

Consistency in parameter rules.....P

No special parameter operations.....T

Exception handling and parallel processing are not explicitly addressed within the language.

The formal parameter and the passed parameter must agree in type according to language rules. However, as was the case in requirement A1, this type checking can be defeated by use of pointers and computed array or table location values.

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

---P---

Dimensions determined at compile time.....T
Size and subscripts can be passed.....F
Type agreement for actual and formal parameters.....P

In general there is agreement of type between formal and actual parameters. Use of computed subscripts and use of pointer capabilities still provide methods for defeating type checking in all cases.

The language can be modified to pass array size and subscripts as arguments and the capability to support more than two dimensional arrays can be included.

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P---

Parameter constants.....T
Parameter variables.....T
Exception conditions.....P
Procedure parameters.....T

The constructs VALUE, LOCATION, and LABEL identify the input, output and exception handling capabilities of the language. Exception control is supported by use of label parameters. There is no formal classification of exception parameters specified within the language.

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

---P---

Optional properties.....P
Fixed at run time.....F

Range and dimension of arrays and tables cannot be specified within procedure definition. All other properties must be specified.

It is not apparent to the reviewer how this requirement can be achieved and the requirements of A1 and A2 simultaneously achieved; perhaps a special construct, GENERIC, to be used only by an elite, tightly controlled group of programmers.

Support of this capability requires a significant change to the syntax of the CORAL66 language.

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

---F---

Variable number of arguments.....F
All but constant number same type.....F
Number fixed at compile time.....T

The number of arguments for all procedures is fixed at compile time. In one sense, passing of a formally declared array name as an argument provides access to a variable number of arguments. This apparently is not the intent of this requirement but could be judiciously used to partially satisfy it.

The syntax of the language can be changed to support this requirement but the magnitude of the change indicates an almost complete redefinition of segments of the language.

D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

---T---

The language provides this capability.

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---P---

Literals of built-in data types.....P
Consistency in value.....P

Floating point numbers cannot be compiled since the compiler works only in fixed point. The existing documentation is unclear as to whether the values of literals are a function of compile time, run time or both.

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---P---

Declare initial values.....P
Allocation scope initialization.....P
No default values.....T

Certain data objects may be initialized when the program is loaded into storage by use of the presetting clause in the data declaration. Presetting is not dynamic, and preset values which are altered by program execution are not reset unless the segment or program is reloaded. An object is not eligible for presetting if it is declared anywhere within

1) the body of a recursive procedure or,
2) an inner block of the program or,
3) an inner block of a procedure body.

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---F---

There is no capability specified within the language for range checking.

The compiler can be expanded to include range checking mechanisms. Checks for compile time usage can be included with relatively little cost. Run time checks, however, will require an extensive overhaul of compiler syntax.

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

---F---

The language does not provide this capability. It can be added as a subset of the potential additions discussed in D4. The language has a very limited array/table capability. Only two dimensional arrays are permitted. There is limited capability for the interrelation of arrays and records. A major restructuring of the language will be necessary to meet the intent of the "TINMAN" requirements.

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

---P---

Shared/recursive substructure capability.....T
Variable/record/array component handling.....T
Typed pointer characteristic.....F
Allocation never wider than access.....F

The pointer capability provided by CORAL66 is wide open. Virtually no checking of type or access scope is possible. Pointers are defined as integers and thus may take on any value permitted an integer. It is possible to access any core location by mistake using the pointer.

In order to satisfy "TINMAN" requirements, several addi-

tions must be made to the syntax of the compiler. First, the pointer must be typed explicitly as such. Next, the rules governing pointer use must be defined. In this case pointers should be limited to record or array structures with a series of restrictions limiting the range of values the pointer may take. These restrictions should preclude computational assignment of value to the pointer. Values should be limited by the compiler to that set which is completely known at compile time.

Obviously this will severely limit the power and usefulness of pointers but stringent type checking and scope checking demand this limitation.

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

---F---

This requirement is not supported by the language. New data types may be introduced implicitly by definition of procedures to perform the function of a new data type. In this case, however, type checking and allocation scope checking rules can easily be violated.

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

---F---

The language does not permit definition of new data types as an operation.

This capability can be added to the language as an extension of E1.

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---T---

CORAL66 completely meets this requirement.

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---F---

The language does not permit new type definition. This capability can be included as part of the extension to type definition.

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class.

A defined type will not automatically inherit the operations of the data with which it is represented.

---F---

The language does not support user defined types.

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

---P---

Enumeration.....F
Cartesian products.....T
Discriminated union.....F
Power set of enumeration type....F

The table mechanism is used to define records. Its use is limited, however. Records do not define types and cannot be used as type constructors.

Arrays, as has been mentioned before, are limited to two dimensions.

The language may be modified to support this requirement. Table and array structures are very limited within the language and should be expanded.

Requirement: E7

Type definitions by free union (i.e., union of nondisjoint types) and subsetting are not desired.

---P---

Does not permit free unions.....F
Does not permit subsetting.....T

Free unions are permitted with use of pointer and overlay functions within the language.

Subsetting is not defined within the language.

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

---F---

* Types cannot be defined within the language.

F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---P---

Initialization of variables is accomplished by a common communicator used by the system. Initialization is performed only once under the rules stated in D3.

Out of scope access to variables is possible when using the pointer facility of the language.

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

---F---

As previously stated, the language does not support definition of new data types.

The language does not distinguish between normal structures and "special" structures. The language can certainly be modified to include such a capability but it will be costly.

Requirement: F3

The scope of identifiers will be wholly determined at compile time.

---T---

This requirement is completely met by the language.

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

---P---

Variety of data and operations.....P
Easily accessible.....T

The language documentation mentions a number of procedures and routines that are available. Whether the libraries contain all things desired by this requirement is not determined.

The cost of providing a "variety" can range from modest to exhorbitant, depending on what functions are considered necessary.

Available routines are easily accessible to the programmer.

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---P---

Accessible at compile time.....T
Other source language routines.....P

The language does not preclude routines written in a different source language. As long as the routines exist in an object form compatible with CORAL66 they can reside in system libraries. There is no provision for interface checking in the language.

Requirement: F6

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---P---

Libraries and compools indistinguishable.....F
Hold anything definable in language.....T
Many levels of access.....T
Many specialized subsets.....P

The segments of a program may communicate with each other through a common global object set and with objects external to the program by means of communicators such as library, external or absolute identifiers as defined in particular implementations.

Subsets may be defined but whether there are "many" is open to individual interpretation.

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---F---

Inclusion of such a capability within the language will depend upon the choice of a standard set of interfaces. Cost of this effort is even more dependent upon this choice.

G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

---P---

Sequential control.....T
Conditional control.....T
Iteration.....T
Recursion.....T
Parallel processing.....F
Exception handling.....P
Asyncronous interrupts.....P

The language provides the following keywords for control mechanisms:
DO
FOR
GOTO
IF
WHILE

Exception handling is provided by use of parameter labels. This does not explicitly provide all the capability required but does provide a modest start for attaining the capability.

The language does not provide an explicit mechanism for interrupt handling. Extra facilities and extensions are available to include the generation of multi-level programs, i.e., those which run under several interrupt levels.

Similarly, there is no explicit definition of parallel processing. Addition of this to the structure of the language will require major language modification.

In general, the language does not explicitly specify how any of the above requirements should be met. Parts of these requirements can be met by use of language capabilities provided for different functions --sort of twisting the intent of the language as it were.

Requirement: G2

The source language will provide a "go to" operation applicable to program labels within its most local scope of definition.

---P--

Go to transfers are to labels defined within the local scope of each program block. Labels are subject to the same rules as variables. Thus the "go to" statement permits jumps from an inner to an outer block or within the same block but not from an outer to an inner block.

Rules governing the "go to" operation can be modified to preclude branches out of scope but then a new mechanism for exception processing must be defined.

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---P---

Based on boolean expression.....T
Based on type from discriminated union.....F
Based on computed choice.....P

An "else" clause is not manditory for all "if-then" expressions. It may or may not be used.

A computed choice for "go to" is provided by the "switch" function. No mention is made of what happens if the switch value is outside the range of statement numbers.

There are no simple mechanisms for common cases.

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g. fixed number of iteration or elements of an array exhausted).

---P---

Termination anywhere in loop.....T
Local control variables.....F
Entry at loop head only.....T
Efficient and clear simple cases.....T
Control value available at termination.....F

Termination within a loop is made possible by several constructs within the language.

Syntax rules do not explicitly state whether loops are entered only at the top.

The control variable may or may not occur within the controlled statement. The controlled variable is a word reference, i.e., either an anonymous reference or a declared word reference. The value of the controlled variable is available for declared word references. Syntax rules explicitly define the order for incrementation.

Syntax for iterative evaluation can be redone to completely meet "TINMAN" requirements.

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---P---

Recursive and nonrecursive routines.....T
Explicitly specify recursive routines.....T
No nested recursive procedures.....U

Language syntax does not state whether procedures may be defined within the body of recursive procedures. If so it is an easy task to restructure the language to prohibit this capability.

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---F---

The language does not explicitly address parallel processing requirements.

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

---F---

CORAL66 does not explicitly include constructs for exception control handling. The only mechanism provided by the language is the capability to pass addresses, as LABEL parameters, for what routine to enable in the event of an exception.

Addition of exception control constructs to the language

will be a costly process.

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

---F---

This requirement appears to expand on the G6 requirements. Parallel processing capabilities can be added to the language. This addition will be costly and may tend to cloud the readability and clarity of the language.

H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

---P---

Free format with statement separator.....T
Mnemonically significant identifiers.....T
Conventional forms.....T
Simple uniform grammar.....T
No special case notations.....P
No abbreviation of keywords or identifiers....T
Unambiguous grammar.....P

Square brackets as used with pointers and switch functions tended to confuse the evaluator occasionally. Also, the differentiation between the uses of square and round brackets was occasionally misleading. These however, are very minor points.

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedence.

---T---

The syntax is completely fixed.

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

---T---

The language satisfies this requirement.

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include

literals for numbers and character strings and a break character for use internal to identifiers and literals.

---P---

Literals for numbers.....T
Character strings.....T
Break character.....F

The language does not contain a break character but can be modified to include one.

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---F---

Multiple lines are permitted within the language structure. It should be very simple to change this feature.

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

---T---

The language defines 42 key words which seems to be few enough.

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

---P---

Single comment convention.....F
Distinguishable from code.....P
Introduced by one or two characters.....F
Contain any charcter.....P
Appear anywhere reasonable.....T
Terminate at end of line.....F
Not prohibit reformatting.....T

Comments may be expressed in two forms; either preceded by the word COMMENT or enclosed in round brackets following a program statement. This can be easily changed to

meet the requirement.

As noted above, more than two letters are required to introduce a comment. Additionally, multiple line comments tend to confuse the reader of complex programs written in the language. These faults can be easily eliminated by modifying the language to conform to "TINMAN" requirements.

Requirement: H8

The language will not permit unmatched parentheses of any kind.

---T---

Requirement: H9

There will be a uniform referent notation.

---T---

The language employs square brackets in referring to members of arrays or tables and round brackets in defining procedure parameters. The complete intent of this requirement may not be met, however, we feel that this is a minor variation.

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---T---

I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic.That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

---T---

This requirement is satisfied by the language.

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---P---

In general, the language does not permit defaults. However, there are some exceptions. For example, recursion is treated as a procedure for compilers lacking the recursion feature. Floating point is treated as integer for compilers not including floating point capability. In all cases of this sort the programmer is advised of defaults.

Requirement: I3

The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---F---

The compiler language does not support explicit specification of the object machine. These capabilities can be added to the language, but at some expense. There is a definite need for an explicit method to specify word sizes, number of bits per word, bit sizes for address formuls, and other machine dependent features.

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile-time variables. In such cases, the conditional will be evaluated at compile-time and only the selected path will be compiled.

---P---

Case statements are not used for conditional evaluation. There are certain procedures available to structure the program to the object machine. For example, floating point operations are defined as integer for those machines with no floating point hardware. Recursive procedures default to procedures on those machines not having the recursion package available. Other options may be available but are not described in the evaluation documentation.

All possible options are compiled. There is no capability to select options for compilation.

Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---T---

The language does indeed provide a simple, clearly identifiable base for use. Currently, the language can be easily extended to increase possible usage. As it now exists, such facilities as bit manipulation, overlay features, floating point processing, recursion, table facility, and fixed number processing can be added or deleted from the language, depending upon the object machine environment.

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

---P---

Available documentation describing compiler limits is precise in some areas and vague in others. For example array dimensions and identifier lengths are explicitly defined. Nesting limits and number of identifiers were not.

The language is designed for use on a number of quite different machines. Limits are probably dependent upon the machine environment in at least some cases.

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---T---

The language satisfies this requirement.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---P---

The objective of the language development has been to permit latitude, not in details but in the presence or absence of major features, which may or may not be considered worth having by the particular installation. A full compiler can provide all features but subsets are available. For example, a floating point feature would not be considered useful to an installation lacking floating point hardware. Therefore, this feature can be eliminated.

Careful use of the language will produce code which provides almost a one- for-one correspondence with machine code.

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

---T---

In so far as we know, translators of the language do not change the effect of the program.

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---T---

The CODE construct provides this capability.

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

---P---

Specify object presentation.....T
Specify time/space tradeoff.....F

This capability is somewhat available through use of the TABLE construct. Field orders, field widths, bit patterns, and field structures can all be specified within a table structure.

There is no mechanism for explicitly specifying tradeoffs. However, the translators do, in fact, optimize data manipulations for the object computer.

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---P---

The open implementation for machine language insertions is clearly met by the language. Procedures are always defined to be closed. There is a macro expansion capability defined within the language, but it does not completely meet the specifications.

CORAL66 EVALUATION SUMMARY

This section presents a summary of the findings of the evaluation of the CORAL66 language. Strengths and weaknesses of the language are presented. Requirements for language modification necessary to meet DOD needs and functions are addressed.

CORAL66 has a set of strengths that specifically meet the requirements of the "TINMAN". These include:

Well defined syntax -- Syntax of the language cannot be changed - Few key words are required to define the language base - No word abbreviations are permitted - Language statements are free format.

Strongly typed -- Identifiers must be fully declared before use.

Block structured -- Scope of identifiers is clearly discernable - Rules for transfers among scopes are well defined and clear.

Feature expandability -- Inclusion of various levels of major features depending upon facility configuration.

Separate compile capability -- Programs may be compiled separately and linked together for execution at run-time.

Machine dependencies excluded from language -- Base language not oriented toward any one or group of machines.

Few defaults permitted -- Defaults limited to capabilities included within specific compiler.

The major weakness identified during the evaluation was the absence of language capability. Basically, CORAL66 supports a very limited subset of DOD "TINMAN" requirements.

Limited control structure capability -- Exception handling capability very limited - Parallel processing not explicitly supported - Interrupt handling not explicitly defined within language.

Small subset of operations possible -- No provision for whole array operations - I/O constructs not available - Power sets not included - Limited Boolean capability.

Data type definition not supported -- No capability within the language to specify new data types.

Evaluation of the language did not identify any capabilities that exceeded "TINMAN" requirements. There were no

special operators or constructs that would provide a programmer with capabilities outside the scope of DOD requirements.

Syntax of the language did conform to the rules specified for the evaluation, i.e., the language syntax would not have to be modified to eliminate clashes with specifications presented within the "TINMAN".

In general, CORAL66, like many other languages of its type, provides a limited number of constructs which can be used to solve several classes of applications. The language contained no unique approaches to problem solving. Rather, the syntax follows the standard, well tested methods. The language is free of machine dependent operations and can be transported to many different machines. As is the case with other languages, specific application programs are implementation dependent and may require many modifications to operate properly on different machines.

Programs written in the language, in many cases, tend to be difficult to follow, due to the comment conventions employed. Maintainability is one of the major DOD goals.

Modification of CORAL66 to more fully support DOD requirements is possible. Some features, such as floating point capability, division with remainder, "NOT" and "NOR" operators, uniform comment notation, and typed pointer key word and syntax for example, can be easily included within the language at relatively little expense. Other features, such as expanded array and record capabilities, user type definition, I/O device definitions, power sets of enumeration, and expanded exception handling for example, will prove to be much more costly, both in terms of time and money.

EVALUATION OF

PASCAL

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Reston, Virginia 22090

November 18, 1976

This report presents an evaluation of the PASCAL language with the requirements listed in the "TINMAN". (DOD Requirements For High Order Computer Programming Languages, "TINMAN" - 1 March, 1976, Section IV). For purposes of comparison PASCAL is considered to be defined by:

The Programming Language Pascal

There are 78 language requirements listed in Section 4 of the "TINMAN". This report compares PASCAL with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The introductory paragraph of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meanings are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DOD "TINMAN" requirements.

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile-time and unalterable at run-time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

---T

Every variable occurring in a statement must be introduced by a variable declaration which associates an identifier and a data type with that variable. The data type explicitly or implicitly defines the set of values which may be assumed by that variable. A data type in PASCAL may be either directly described in the variable declaration, or it may be described by a type identifier. In the latter case the type identifier must be described by an explicit type declaration.

Requirement: A2

The language will provide data types for integer, real (floating point and fixed-point), Boolean, and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type).

---P

Integer	---T
Floating point	---T
Fixed point	---F
Boolean	---T
Character	---T
Arrays	---T
Records	---T

The values of integers are the integers within a range which are implementation dependent.

The values of floating point real quantities are a subset of real

numbers which are implementation dependent.

The values of Boolean variables are denoted by the keywords "true" and "false".

There are two character types: char and alfa. The set of values of the type char is the character set available on the printers of a particular installation. Alfa type values consist of sequences of characters whose length is defined as being implementation dependent and normally infers the number of characters packed per word on the target machine. Individual characters are not directly accessible, but alfa quantities can be unpacked from an alfa variable to a char array by a standard procedure.

Both char and alfa quantities are denoted by themselves enclosed within quotes.

In an array structure, all components are of the same type. A component is selected by an array selector, or computable index, whose type is indicated in the array type definition and which must be scalar. It is usually a programmer defined scalar type, or a subrange of the type integer.

In a record structure, the components (called fields) are not necessarily of the same type. In order that the type of a selected component be evident from the source program text at compile-time, a record selector does not contain a computable value, but consists of an identifier uniquely denoting the component to be selected. These component identifiers are defined in the record definition.

Fixed-point data types are not constituents of PASCAL nor can they be constructed from the primitive data types unless a subrange of the type real, that is with specified upper and lower bounds, is considered valid. The addition of fixed-point data types to PASCAL as either a primitive data type or as a subset of the real data type by incorporating a precision clause within the subrange declaration would appear to be a straightforward extension to the language.

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. The specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---F

PASCAL does not support the concept of precision specification of floating point variables either at a block structure level or on an individual variable basis. It would be a straightforward proposition to extend the language to require that each block in which floating point variables are declared include a declaration of the precision required for arithmetic operations performed within that block and for the actual data representations of local floating point variables. However, it is anticipated that to allow unique precision specifications for each variable would result in possibly inefficient object code and an excessive compile-time burden resulting from increased subtype checking and the inclusion of object code for data representation conversion. Alternatively the precision could be utilized by the compiler only to determine if the target machine were capable of supporting the specified precision.

Requirement: A4

Fixed-point numbers will be treated as exact quantities which have a range and fractional step size which are determined by the user at compile-time. Scale factor management will be done by the compiler.

---N/A

Since PASCAL does not include a real fixed-point data type, either as a primitive or the ability to generate it from its primitives, then the question of fixed-point range and resolution are not applicable. However, if the extension concerning the introduction of real fixed-point data types described in the response to Section A2 were implemented as a primitive data type, then the resolution and lower and upper bounds could be specified as an extension to the subrange clause. The information within such a clause would provide all the attributes necessary for the compiler to handle scale management and produce object code for bound checking, if desired.

Requirement: A5

Character sets will be treated as any other enumeration type.

---P+

The internal representation of PASCAL char and alfa type data items is not defined within the language and would most probably be fixed for any given implementation. However, the ability to define scalar types will allow any desired coallation sequence for an alphabet. As a result several alphabets, including ASCII and EBCDIC, could be defined, thus, mapping from one to another would be a trivial task. The ASCII and EBCDIC alphabets are not predefined within the language specification, thus, requiring each user to declare them according to the definable scalar format.

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array. The number of dimensions, the type of lower subscript bound will be determinable at compile-time. The upper subscript bound will be determinable at entry to the array allocation scope.

---P+

An array type is a structure consisting of a fixed number of components which are all of the same type, referred to as the component type. The elements of the array are designated by indices, values belonging to the so-called index type. The array type definition specifies both the component type and the index type. Both lower and upper bounds are normally specified at compile-time. However, utilization of the class type definition in conjunction with standard procedure allocation provides the ability to allocate the array at scope entry. This is done dynamically, which is in conflict with the "TINMAN" requirement but the class type definition requires that the maximum array size be explicitly specified at compile-time. It is our opinion that this requirement satisfies the intent of the "TINMAN" requirement.

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile-time. The name and type of each record component will be specified by the user at compile-time.

---P+

A record is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, its type and the identifier which denotes it. A record type may have one or more alternative structures, referred to as variants, as specified by the "TINMAN". However, at a given time, the specific variant is selected by the value of a single fixed field of the record, referred to as the tag field. This is in conflict with the "TINMAN" requirement which requires that alternative structures be selected by a general Boolean expression. The extension of the language to satisfy this subrequirement is considered to be extensive since it would imply a totally different selection mechanism than the currently specified tag field.

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or a union type containing that type. Reference will retrieve the last assigned value.

---P+

The PASCAL assignment statement will allow a variable of any given type, with the exception of class and file types to be assigned to an expression of the identical type. Thus, an entire array can be equated to an equivalent array, an entire record to an equivalent record, or a component of a record to a similar equivalent component or primitive variable of identical type. There are two relaxations to this general rule of type consistency:

1) Where the type of the dependent variable is real and the type of the expression is integer or a subrange thereof.

2) Where the type of the expression is a subrange of the independent variable.

These type checking relaxations could readily be deleted from any given implementation, however, the ability to assign file and class type data items would be considered a major impact and would require a detailed analysis to determine any undesirable side effects or conflicts.

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---P+

The PASCAL language identity operators for equality (=) and inequality (≠) may be used with operands of any type with the exception of class and file types, the result is always type Boolean.

To include the operand types class and file would be considered a major impact for the same reasons as described in the response to Section B1 of this report.

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---P+

The relational operators defined for PASCAL are as follows:

$$= \quad \neq$$

$$< \quad >$$

$$\leqslant \quad \geqslant$$

The identity operators (=,≠) may be used with any data type with the exceptions of those noted in Section B2. The relative operators (<,>,⩽,⩾) may be utilized with any scalar or subrange type. The scalar types include the predefined items, integer, real, Boolean, char and alfa, and those defined by enumeration. It should be noted here that the semantics of the relative operators within the alfa context are not obvious and are defined below.

If X and Y are alfa identifiers, the value of which consist of sequences of n characters, where n is an implementation dependent parameter, (see Section A2) then:

X<Y if Xi=Yi for 1<i<k-1 and Xk<Yk

X>Y if Xi=Yi for 1<i<k-1 and Xk>Yk

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed-point arguments and remainder), and negation.

---P+

The arithmetic operators of PASCAL are defined as follows:

*	multiplication
/	division
div	division with truncation
mod	m mod n=m-((m div n)*n)
+	addition or identity
-	subtraction or inversion

The multiplication (*), division (/), addition (+), and subtraction (-) operators permit the operands to be real, integer, or mixed. The result for multiplication, addition, and subtraction is integer only if both operands are integer, else it is real. The result for the division operator is always real. The div and mod operators may be utilized with integer operands only, yielding integer results of a truncated integer division and remainder, respectively. There is no exponentiation operator. To add that feature would be relatively minor if a decision concerning the semantics and result type could be resolved. For example, the expression A**X**Y if evaluated with a conventional left to right precedence is equivalent to A**(X+Y) which some would argue, is not what was intended. If evaluated with a right to left precendence, the result is equivalent to A***(X**Y). This results in the relatively complex semantic rules that in the absence of parenthesis expression operators are evaluated according to precedence and left to right with the exception of the exponentiation operator which is evaluated right to left.

In addition, the result type of an exponentiation operation, when both operands are integer also causes considerable confusion unless clearly defined. Finally, the possibility of complex results introduces a further confusion factor. The designer(s)

of PASCAL were perhaps wise in omitting the exponentiation operator from the arithmetic operator set.

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will be on the least significant digits and will never be implicit for integers and fixed-point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

---P

No truncation of most significant digits ---U

No implicit truncation or rounding for fixed-point and integer operands ---T

Implicit rounding beyond specified precision ---N/A

The source material used for this report was addressed to the PASCAL language definition, not a compiler implementation, thus, it was not possible to determine if most significant digit truncation would ever occur. Stating that it should never occur is an incomplete specification, since there are obvious instances where it could occur, the most common, perhaps, is the assignment of a floating point expression to an integer or fixed-point operand. Since the latter will generally have a more restricted range then should that range be exceeded what action is to be taken? In most environments the choice is to either ignore the discrepancy resulting in what is essentially modulus arithmetic or to abort the execution of the program and return control to the system executive. It would certainly be feasible and, perhaps, desirable to the HOL project to provide the source language with a mechanism to specify those actions such that the user would retain control, if desired. This could be accomplished by optionally specifying at the entrance to each block an identifying procedure label which would be executed if overflow occurs. Within that procedure, the user would determine the need to either abort program execution or to continue.

Implicit truncation cannot occur within PASCAL assignments since a real expression cannot be assigned to an integer operand. In addition, within an expression, mixed mode operations always result in a real value. Explicit truncation of real expressions can be achieved by use of the standard function, trunc.

Since floating point precision is not an attribute of the

language, implicit rounding beyond a specified precision is not applicable. However, if precision specification were included as described in Section A3 then it is anticipated that a real expression would be evaluated within the maximum precision supported by the target machine, only the operation of assignment would round the result to the specified precision.

Requirement: B6

The built-in Boolean operations will include "AND", "OR", "NOT" and "NOR". The operations on scalars will be evaluated in short circuit mode.

---P

The built-in Boolean operations of PASCAL are as follows:

$\wedge$(logical "AND")

$\vee$(logical "OR")
$\neg$(logical "NOT")

The operation "NOR" is not explicitly available within the language. It is anticipated that it could be incorporated with a minimum effort. However, purists may argue that it is not only unnecessary but undesirable since both "NOR" and "AND" can be represented by use of the "NOT" operator applied to a subexpression in parentheses.

The language definition does not specify the semantics of short-circuit evaluation, however, any implementation could of course, employ short-circuit evaluation techniques.

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---T

Scalar operations on arrays ---T(U)

Assignment between records and arrays of comformable type ---T

It is clear from the PASCAL language definition that assignments between records and arrays of conformable type are valid. In addition, entire variables representing arrays can apparently be operands of expressions. However, the semantics of such operations is undefined, thus, multiplication of arrays could imply matrix multiplication. The obvious solution to this would be to include explicit semantic definitions within the language specifications.

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels, or devices (including terminals). These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---p+

A file structure in PASCAL is a sequence of components of the same type. A natural ordering of the components is defined through the sequence. At any *instant*, only one component is directly accessible. The other components are made accessible through the standard file positioning procedures: put, get and reset. At any given time, a file is in one of three modes called input, output and neutral. According to the mode, a file can be read sequentially or it can be written to by appending components to the existing sequence of components. File positioning procedures may determine the current mode. The file type definition does not determine the number of components, and this number is variable during the execution of the program. The components of a file may be a simple primitive data type (i.e., integer) or may be complex structures, the specific form of the structures being defined by a record definition. Two standard file variables can be considered as predeclared for all implementations as:

```
input, output: file of char
```

The file input is restricted to input mode (reading only), and the file output is restricted to output mode (writing only). A PASCAL program should be regarded as a procedure with these two variables as formal parameters. The corresponding actual parameters are expected either to be standard input and output devices of the target computer installation or to be assignable by the system commands of the target computer. As a result, it would appear that files would not normally be dynamically assignable, however, additional standard procedures could be added to a given

implementation to provide a dynamic reassignment capability.

The standard file positioning procedures are defined below:

put (F) Advances the file pointer of file F to the next file component. It is only applicable if the file is in either the output or neutral mode. After execution, the file is in the output mode.

get (F) Advances the file pointer of file F to the next file component. It is only applicable if the file is in either the input or neutral mode. If there does not exist a next file component, the end of file condition arises, the value of the variable denoted by the file pointer becomes undefined, and the file is put into the neutral mode.

reset (F) The file pointer of file F is reset to its beginning, and the file is put into the neutral mode.

N.B. Initially a file variable is in the neutral mode.

In addition to the file positioning procedures, a further standard function is available to determine the end-of-file status of a file. The function is eof with the file variable as an argument. The function type is Boolean which when true, indicates an end-of-file status. Any other conditions, such as data transfer error, are not available as standard functions but could be included in any given implementation at an anticipated low to moderate cost.

Requirement: B11

The language will provide operations on data types as power sets of enumeration types (Section B6). The operations will include union intersection, difference, complement, and an element predicate.

---P

A PASCAL powerset type defines a range of values as the powerset of another scalar type, the so-called base type operators applicable to all power set types are:

union

* intersection

\- set difference

••• membership

----------P

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left to right.

---T

The source documentation specifies that sequences of operators of the same precedence level are executed from left to right. The rules of precedence are reflected by the specified syntax definition.

Requirement: C2

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---T

Expressions are constructs denoting rules for obtaining values of variables and generating new values by the application of operators. Expressions consist of operands (i.e., variables, constants and functions) and operators.

The rules of composition specify operator precedences according to four classes of operators. The Boolean negation operator () has the highest, multiplying operators (*,/,div,mod,^), then the so-called adding operators (+,-,V), and finally, with the lowest precedence, the relational operators (=,≠,<,>,≤,≥). These precedence levels are fixed and cannot be altered by the user as required by the "TINMAN". In addition, conventional explicit parenthetical pairs can be included within expressions, to any desired level, to specify the intended evaluation sequence or to ensure the required evaluation order of operators of the same precedence level.

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

---T

Requirement: C4

Constant expressions will be allowed in programs anywhere constants are allowed, and constants will be evaluated before runtime.

---F

PASCAL does not support the concept of constant compile-time expressions either in the form of literal or parameterized constants. Parameterized constants are definable and may appear at any point where a literal constant is specified. Parameterized constants are generally favored, since they allow a single assignment of a compile-time identifier to a value which may subsequently be utilized by numerous references. Coupled with the capability of compile-time expression evaluation, compile-time identifiers may be functions of other similar identifiers, thus minimizing programmer errors and providing a readily maintainable environment. The addition of such a capability to PASCAL would be considered a moderate effort.

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, types, exception handling, parallel processes, declaration, or built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters.

---T

Consistency has been achieved for parameter handling within PASCAL, primarily inherited from its predecessor ALGOL 60. Identifier references, be they actual or formal, are fully typed to ensure that consistency.

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at

compile-time. The size and subscript range for array parameters need not be determinable at compile-time but can be passed as a part of the parameter.

---P

Actual and formal parameters will agree in type.
---T

Rank of parameter arrays is fixed at compile-time.
---F

Parameter array upper and lower bounds can be passed.
---T

Actual parameters are, of course, defined by the normal data declaration constructs. Formal parameters are declared, or more correctly specified, by appending the appropriate clause to the formal parameter synonym. There exist four clauses to specify the four kinds of parameters: variable, constant, procedure, and function.

In the case of variable-parameters the actual parameter must be a variable. If it is a variable denoting a component of a structured variable, the selector is evaluated prior to execution of the procedure. If the parameter is a constant parameter, the corresponding actual parameter must be an expression. If the parameter is a procedure or function, the corresponding actual parameter must be a procedure or function identifier, respectively. Actual to formal parameter correspondence is achieved by the ordinal positions of the parameters in the lists of actual and formal parameters and bounds.

The rank of arrays are not specified by the formal parameter declarations and, thus, must be passed as a part of the actual parameter upon invocation of the procedure.

Requirement: C7

There will be only four classes of formal parameters. For data there will be those acting as constants representing the actual parameter value at the time of call and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P+

Call by value parameters ---T

Call by name parameters ---T

Exception handling parameters ---U

Procedure identifier parameters ---T

The types of procedure parameters available to PASCAL are defined in Section C6 of this report. They are: constant, variable, procedure, and function. Constant and variable parameters are equivalent to "call by value" and "call by name" parameters, respectively. The procedure identifier parameter allows procedures to be passed as actual parameters to other procedures as mandated by the "TINMAN".

It is unclear from the source documentation as to the availability of exception handling parameters, most commonly encountered, in the form of labels.

Requirement: C8

Specification of the type, range, precision, dimension, scale, and format of parameters will be optional in the procedure declaration. None of them will be alterable at run-time.

---P-

Specified attributes optional ---F

Attributes unalterable at run-time ---T

All procedure formal parameters are fully typed within the procedure heading to ensure consistency of type with the actual parameters encountered at procedure invocation. If those formal parameter attributes were optional, it is difficult to envision how type consistency could be effectively managed. In fact, this requirement appears to be in conflict with much of the "TINMAN" since overall data type consistency is one of the major conventions dictated.

Requirement: C9

There will be provisions for variable numbers of arguments, but

in such cases, all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile-time .

---F

PASCAL procedure parameters are limited to the fixed number specified by the declaration. Each procedure invocation must, of course, include the same number of actual parameters.

The concept of variable numbers of arguments to procedures is, thus, somewhat foreign to this language. However, within the specified restraints, it would appear to be a realistic requirement. The impact of including the capability within a given language implementation would be moderate to major.

Requirement: D1

The user will have the ability to associate constant values on any type with identifiers.

---T

The constant definition facility provided enables an identifier to be a synonym for a constant. The constant may be a signed or unsigned real, or integer number, a character string, a user defined identifier, or the undefined pointer constant nil.

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---T,N/A

Consistent literal syntax and interpretation
---T

Equivalent literal values for programs and data
---N/A

There is a single syntactic definition and hence implied interpretation for references to all literals thus providing a consistent notation.

The requirement for equivalent literal value representation in both program and data input is considered out of scope of the language definition which does not address the run-time environment.

Requirement: D3

The language will permit the user to specify the initial values of individual variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---F

PASCAL does not support the concept of variable data initialization. The value of variables of entry of allocation scope is undefined. Since, for efficient memory management it is normal for noninterfering program blocks to share data space, initialization cannot occur at either compile or load time and must become a dynamic run-time feature, this is in partial conflict with the "TINMAN" requirement.

The extensions to the language to accomodate initialization would be minor but the impact on data management to fully satisfy the requirement would be moderate.

Requirement: D4

The source language will require its users to specify, individually, the range of all numeric variables and the step size for fixed-point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---F

Numeric range specification mandatory
---F

Fixed point step size specifications mandatory
---N/A

Range and step size specifications do not define a new data type
---F

PASCAL does support the concept of range specification for both the predefined scalar quantities (integer, etc.) and those data types defined by enumeration. However, contrary to the "TINMAN" requirement such range specifications are optional and each range specification is considered a unique data type. To modify the language to require that all numeric variables require a range specification would require minimal effort, but the conceptual change of not considering such subrange types to be unique would be major, since it would change the total structure of the language.

Perhaps, the object of the requirement is to allow assignments, etc., between variously ranged variables of the same general class without violating the stringent type checking rules specified within the "TINMAN". As previously described within this report, PASCAL conveniently skirts this issue by relaxing type checking between scalars and subranges thereof. If this approach is considered acceptable, the overall impact would be minimal.

Since fixed-point variables are not supported by PASCAL, the question of step size is not applicable. However, if fixed-point variables were introduced, it would appear that for total consistency and for little additional effort, the step size attribute could be introduced.

Requirement: D5

The range of values which can be associated with a variable, array or record component will be any built in type, any defined type or a contiguous subsequence of any enumeration type.

---T

PASCAL data declarations are recursive in that element values of all identifier types can themselves be typed identifiers. This permits arrays to be components of records or arrays and permits records to be components of records or arrays. An element value may be a contiguous subsequence of any enumeration type if that subsequence is defined as a subrange type of the entire sequence.

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and for recursive substructure. The pointer property will only affect the use of variables (including array and record components) of same data types. Pointer variables will be as safe in their use as are any other variables.

---P+

Pointer mechanism(s) available ---T

Shared data structures ---T

Recursive data structures ---T

Pointer property type is bound to data structure type ---T

Pointer property will be only for variables of composite components and of composite array and record types ---F

Allocation scope wider than access scope ---U

Type and access restricted pointers ---T

PASCAL 'class' variables which are dynamically assigned by reference to the standard procedure allocation may only be referenced by pointer type variables. Class variables may be of any type, either user defined or predefined scalar, which is in conflict with the "TINMAN" requirement which dictates that only composite data structures may be referenced by pointers. A pointer is bound to a given 'class' variable by an appropriate declaration and, thus, ensures type checking for pointer variables in a fashion compatible with nonpointer variables. The limiting of be a minor change to the language. Since more than one pointer may be bound to a single 'class' variable shared data structures can be achieved.

It is clear that since 'class' variables are dynamically allocated, recursive data structures are feasible. However, the source documentation does not specify if and how such data space is deallocated. Since allocation is explicit but must be within the bounds of the pointer scope it would appear that references could occur prior to the initial allocation unless compiler topographical flow analysis is included or run-time checks are performed.

Requirement: E1

The user of the language will be able to define new data types and operations within programs.

-----P+

The extensive user data declaration facilities fully satisfies the "TINMAN" requirement for defining new data types.

Operations can be defined, only by functions or procedures which do not provide a means of defining infix operators. The requirement to define new infix operators would appear to be in conflict with H2 which specifies that "new infix operator precedence" shall not be definable. If new infix operators are to be defined, both the semantics and precedence of those operators would require definition unless the definition were described in terms of existing operators in which case the precedence is implied.

Requirement: E2

The "use" of defined types will be indistinguishable from built in types.

---T

The syntactical structure of PASCAL for reference to any identifier or constant, be it a built-in or user defined type is identical thus satisfying this requirement. The semantics of a statement referencing data types is a function of the general class of data types (e.g., scalars) be they built-in or user defined. For example a relational test of scalar identifiers has both the same syntax and semantics if the identifiers represent integers or those defined by enumeration.

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---P+

The base language provides all the elementary components from which more complex components may be constructed. The constructions may exist in the program, or if the specific implementation allows, in a library.

In general, there are no default declarations within the language with the exception of the constant specificator, const, associated with formal parameter delcarations. To require this keyword would be a trivial effort.

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---p

The operand types with which the defined set of operators can be applied is specified and it is not possible to extend those operators to additional variable types. However, since user defined variable types fall into a given set of general classes (scalars, powersets, etc.), the appropriate existing operators for the operands general class are applicable.

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the type and set of operations applicable to the class. A defined type will not automatically inherit the operations of the data with which it is represented.

---p

User defined variable types, as explained in E4, fall into one of several classifications. The operations pertinent to the general type are inherited and cannot be modified by the user declaration.

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the powerset of an enumeration type. These definitions will be processed entirely at compile-time.

---P+

Definable by enumeration ---T

Definable by cartesian products ---T

Definable by discriminated union ---F

Definable by the powerset of an enumeration type ---T

The basic definable data types are the scalar types. Their definition indicates an ordered set of values, i.e., introduces an identifier as a literal constant representing each value in the enumeration set.

Structured data types are described by describing the types of their components and by indicating a structuring method. Both array and record structure are construed as definitions by cartesian products since they provide definitions in terms of other in built or user defined types.

Powerset definition is explicitly available for scalar base types, either those defined by enumeration or the predefined types (integer, etc.).

Definition by union of disjoint sets is not included though such sets could, of course, explicitly be defined by enumeration of all of the literal elements from the desired disjoint set. Alternatively the superset could by initially defined and subsets defined by the subrange types feature. With either approach, it would be the users' responsibility to ensure that the subsets were disjoint.

To add a language construct to provide disjoint union, typing would be considered relatively minor.

Requirement: E7

Type definitions by free union (i.e., union of nondisjoint types) and subsetting are not desired.

---p-

Type definitions by free-union are not available in PASCAL. Subranging of scalar types is permitted and such subranges, contrary to the "TINMAN" requirement, are construed as new data types. To alter these semantics to satisfy the requirement would be considered a moderate to major task.

Requirement: F8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at time of allocation and deallocation of variable of the type.

---p-

n. The source documentation used for this report did not specify if and how allocated data is deallocated. However, it is apparently not explicit and thus finalizatin actions cannot be specified.

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---- F ----

The PASCAL block structure dictates that the access and allocation scopes are identical with the exception of the class type variables. Since the allocation of class variables is dynamic and explicit, unless a given compiler implementation includes topographical flow analysis, it would be possible for a user to attempt to access a class variable, via a pointer, outside of the allocation scope of that variable.

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

The requirement is partially met, as described in F1 above.

---P

Requirement: F3

The scope of identifiers will be wholly determined at compile-time.

---- T ----

All identifiers must be defined. Access to an identifier is guaranteed within the scope of declaration. Identifiers may be redefined either within imbedded scopes, in which case references apply to the most deeply imbedded declaration scope surrounding the reference or within disjoint scopes. These rules ensure that the scope of all identifiers is a function of the program's static structure and are, thus, determinable at compile-time.

Requirement: F4

A variety of application oriented data and operations will be available in libraries and easily accessible in the language.

---- p ----

The PASCAL language does not provide the mechanism for including compile-time library files in which application oriented data structures could be defined. It is anticipated that for any given implementation, data declaraton files could be readily merged either by an extension to the language or a preprocessor.

The following operations are, however, assumed to be predeclared in every implementation of PASCAL:

put (f)	File positioning procedures.
get (f)	
reset (f)	
alloc (p)	Allocate an additional component to a class variable.
pack (a, i, z)	Pack a character array into an alfa variable.
unpack (z, a, i)	Unpack an alfa variable into a character array.
abs (x)	Arithmetic value for arithmetic value.
sqr (x)	Arithmetic function for squaring.
sin (x)	Trigonometric functions.
cos (x)	
arctan (x)	Inverse trigonometric function.
ln (x)	Natural logorithmic function.
exp (x)	Exponential function.
odd (x)	Boolean function for determining x mod 2=1.
eof (f)	Boolean function for sensing end-of-file.
trunc (x)	Arithmetic function for obtaining integepart of a real quantity.
ord (x)	Arithmetic function for determining the ordinal position of a character in its defined set.
chr (x)	Character function for determining the character whose ordinal position in the

set is x.

succ (x) — Scalar function for finding the successor value of x from its set.

pred (x) — Scalar function for finding the predecessor value of x from its set.

Further the language specifies that any implementation may feature additional predeclared functions or procedures.

Requirement: F5

Program components defined within the current program and not in the base language will be maintained in compile-time libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---- F ----

The PASCAL language definition does not address the concept of externally defined modules in any form, since such features have traditionally been considered functions of the compiler environment and are frequently implementation dependent. However, it is believed that such an extension could readily be appended to the PASCAL language for a moderate effort.

Requirement: F6

Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---- F ----

The language definition, as with Requirement F5, does not address compiler and run-time environments nor any references to items external to the current program. In the practical situation, of course, it is desirable to provide the facility to include items defined within compools and to explicitly reference library items. To ensure the integrity of type checking would require, at least for library references, the maintaining of compile-time data in addition to the object code.

The additions to the language to enable references to compool and library elements is almost trivial, however, to estimate the impact on a given implementation would be futile because of the environment dependency.

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---- P ----

In addition to the specified standard I/O procedures for accessing files, the language definition provides for an extended set of standard procedures. This feature could be readily utilized the capabilities for this requirement. However, it would be desirable to include, as part of the language definition, the identifiers and protocol of those procedures to ensure that standardization could be achieved. The effort to do this cannot, of course, be estimated until the specific functions required are defined.

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structure for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

---- P ----

Sequential control . . . . . T

Conditional control . . . . . T

Iterative control . . . . . T

Recursive control . . . . . T

Parallel processing . . . . . F

Exception handling . . . . . F

Asynchronous interrupt . . . . . F

The language elements which provide the above requirements are identified below:

IF: The IF statement specifies that a statement be executed only if the specified Boolean expression is true. If it is false, no statement is to be executed or the alternate statement following the connective ELSE is to be executed.

CASE: The CASE statement consists of an expression (the selector) and a list of statements, each labeled by a constant of the type of the selector. It specifies that one statement be executed whose label is equal to the current value of the selector.

WHILE: The WHILE statement specifies that a statement be repeatedly executed while the value of a Boolean expression is true. If its value is false at the beginning, the statement is not executed at all.

REPEAT: The REPEAT statement specifies that the sequence of statements between the symbols REPEAT and UNTIL are repeatedly (and at least once) executed until the value of a Boolean expression becomes true.

FOR: The FOR statement indicates that a statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the control variable for that statement. Expressions are included to specify the initial and final values for the control variable. At each iteration, the value of the control variable is replaced by either the successor or predecessor of the current control variable

value. The choice is determined by use of the alternate symbols TO or DOWN TO which separates the initial and final value expressions.

GO TO: The GO TO statement serves to indicate that further processing should continue at another part of the program text, namely at the place of the label. The transfer of control is unconditional.

The scope of access of a label is limited to the structure in which it is defined, and all structures defined within it, with the exception of procedure and function defintions. In those instances, it is invalid to branch out of or into.

Recursive structures are available in the form of procedure declarations where only direct recursion is valid. This property is implied by a reference to a procedure identifier within the executable test of the same procedure.

Parallel processing is not explicitly available as a structure of PASCAL. If it were to be included on a limited basis by the introduction of definable tasks, in a fashion syntactically similar to procedures and if identifier references were limited to local declarations and formal parameters, the effort would be moderate. If capabilities beyond this were required, extensive consideration of the protocol of data handling would need to be defined.

Similarly, interrupt handling is not explicitly defined though they could be processed by the inclusion of further standard procedures, and such a method may prove satisfactory if agreement concerning the semantics of such procedures could be reached.

Requirement: G2

The source language will provide a "GO TO" operation applicable within its most local scope of definition.

---- P ----

The PASCAL GO TO statement does not limit the referenced label to the most local scope. The labels obey the rules of all identifiers, in that the scope of access includes the structure in which it is defined and all structures declared within the structure of definition.

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---- P ----

Based on Boolean expression . . . . . T

Based on subtype from discriminated union . . . . T/U

Based on computed choice . . . . . T

The IF statement provides the primary control structure and dictates that the condition be a Boolean expression as required by the "TINMAN". Data types defined by enumeraton and as powersets thereof may utilize the IN operator which may be construed as a discriminated union but returns a Boolean value. The CASE statement provides conditional statement execution based on the value of a variable of any type. No explicit action is defined for a CASE statement should the value of the expression be out of the bounds specified by the CASE statement.

The IF statement is not fully partitioned, in that the ELSE substructure is optional. However, the semantics of the following statement are unambiguously defined:

```
IF <expression-1> THEN IF <expression-2> THEN <statement-1>
                                         ELSE <statement-2>
```

by interpreting the construct as equivalent to:

```
IF <expression-1> THEN
                  BEGIN IF <expression-2> THEN <statement-1>
                        ELSE <statement-2>
                  END
```

It would appear that the intent of the "TINMAN" requirement is met though it requires a semantic definition to qualify the ambiguous syntax.

Requirement: G4

The iterative control structure will permit the termination condition anywhere in the loop, require control variables to be local to the iterative control, allow entry only at the head of the loop, and not impose excessive overhead in clarity or run-time execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

---- P+ ----

Termination anywhere within loop . . . . . P

Local control variables . . . . . F

Entry at loop head only . . . . . T

Efficient and clear simple cases . . . . . T

Control value available at termination . . . . . U

The iterative structures WHILE, REPEAT, and FOR are fully described in G1 and satisfy the general intent of the "TINMAN" requirement. The WHILE statement provides termination at the head of a loop while the REPEAT statement provides termination at the end of the loop. In addition, exit may be achieved from any point within the loop by combining an IF statement with the unconditional GO TO. However, use of such a technique does not limit program execution to the statement following the iterative structure as is, perhaps, intended by this requirement.

The control variable of a FOR statement, which provides for a predetermined number of iterations, is not local to iterative structure and its value at termination is undefined, unless termination is a result of a branch (GO TO) out of the structure. The only restriction concerning the control variable is that it may not be assigned a value within the structure.

The impact of fully satisfying this requirement would require careful consideration of the total language, since it may violate the general rules concerning identifier scope. As a result, the anticipated effort involved was not predicted.

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---- P ----

Recursive and nonrecursive routines . . . . . T

Explicit specification of recursive routines . . . F

No procedure definitions within recursive routines F

PASCAL procedures may be recursive, but apparently only direct

recursion is valid, since that property is implied only be reference to a procedure identifier within the same procedure declaration. Recursive procedures are not explicitly declared. In addition, all identifiers must be declared prior to reference, thus, precluding indirect recursion at any level. A further violation of the "TINMAN" requirement is that procedure declarations are valid within more global procedure declarations be they recursive or otherwise.

As a result of the somewhat unclear language definition concerning indirect recursion and the uncertainty of the "TINMAN" on this matter, it is difficult to make an assessment on overall impact. However, it is safe to assume that at best it would be moderate.

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---- F ----

The language does not explicitly provide for parallel processing, pseudo or otherwise. This item was discussed in Section G1 of this report and, although any extension of the language to satisfy this requirement would be a major impact, it could be minimized if constraints described in G1 were provided. The net result of such an extension would only partially satisfy this requirement.

Requirement: G7

The exception handling control structure will permit the user to cause transfer and control of data for any error or exception situation which may occur in a program.

---- P ----

PASCAL does not specifically address exception handling as there are no control constructs dedicated to that function. However, extensions to the nebulous 'standard' procedures and the ability to pass procedure identifiers as actual parameters to procedure invocations provides an exception handling capability. The protocol of this requirement is, thus, not dictated by the language

and would require rigorous definition for a given system.

Addition of formally defined exception handling constructs would be an extensive effort.

Requirement: G8

There will be source language features permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, give access to real time clocks, permit asynchronous hardware interrupts to be treated as any other exception situation.

---- F ----

This requirement appears to be a consolidation of G1, G6, and G7 addressing interrupt handling, parallel processing, exception handling, etc. As previously described within those sections, it is feasible that they could be partially satisfied by extensions of existing capabilities. However, since PASCAL does not specifically address real time processing it is probable that the extensions to satisfy these requirements should take the form of new constructs to the language. It would be necessary to undertake a detailed analysis of the language to ensure that the ground rules of the current language are not violated nor are seriously overlapping functions created. These criteria may seem obvious but an examinatton of other languages reveals that inconsistencies, incomplete definitions, and functional overlap are all too often encountered. The impact on the language to satisfy this requirement would, no doubt, be major but without a detailed analysis, which is considered beyond the scope of this report, an accurate estimate cannot be made.

Requirement: H1

The source language will be free format with an explicit statement separator, allow the use of mnemonically significant identifiers, be based on conventional forms, have a simple uniform and easily parsed grammar, not provide unique notations for special cases, not permit abbreviation of identifiers or keywords, and be syntactically unambiguous.

---- P+ ----

Free format with statement separator . . . . . P+

Mnemonically significant identifiers . . . . . T

Conventional forms . . . . . T

No special case notations . . . . . T

No abbreviations of keywords or identifiers . . . T

Unambiguous grammar . . . . . P+

This requirement is almost entirely satisfied. The discrepancies appear below:

The source language is totally free format and does not require a statement separator. This feature is achieved by the simplicity of the syntax.

There appears to be only a single syntactical ambiguity, to this evaluator, concerning the optional ELSE clause of an IF statement as described in Section G3 of this report. The ambiguity is resolved by a precise semantic definition. Alternatively the deficiency could be solved by defining the ELSE clause as mandatory and introducing a null statement into the language. This latter feature would be required as a result of lack of statement separator.

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierachies, introduce new precedence rules, define new keyword forms, or define new operator precedence.

---- T ----

The syntax and semantics of PASCAL are completely defined and can

neither be altered nor extended.

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 ASCII character subset.

---- P ----

Although the required character set is considered suitable for publication, the 89 unique characters preclude the use of an ASCII subset. Reduction of the character set could be reduced to conform to that requirement by restricting alphabetic characters to a single case.

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

---- P+ ----

Numeric literals . . . . . T

Character string literals . . . . . T

Break character . . . . . F

A break character is not included witnin the valid character set but could readily be added, however, since the language is totally free form it does not appear to be necessary.

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---- F ----

The free format of PASCAL precludes the recognition of lines within the source input. To include object characters in character strings is feasible and would be a trivial change.

Requirement: H6

Key words will be reserved, be very few in number, be informative, and not be usable in contexts where an identifier can be used.

---- T ----

PASCAL contains only 30 reserved keywords which is considered very few for a language of this power. The predefined procedure identifiers are not included in this count.

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, be introduced by a single (or possibly two) language defined characters, permit any combination of characters to appear, be able to appear anywhere reasonable in programs, automatically terminate at end of line if not otherwise terminated, and not prohibit automatic reformatting of programs.

---- P+ ----

Single comment convention . . . . . T

Distinguishable from code . . . . . P

Introduced by one or two characters . . . . . T

Contain any character . . . . . P+

Appear anywhere reasonable . . . . . T

Terminate of end-of-line . . . . . F

Not prohibit reformatting . . . . . T

Comments in the PASCAL language may appear any two lexical units (i.e., identifiers, numbers, operators, etc.), are introduced and

terminated by unique symbols, {and} respectively, thus precluding the use of the terminating symbol, }, as a comment character. As a result, it is probable that to a human reader comments are not "easily distinguishable from code". This subrequirement appears to conflict with that requiring "an combination of characters to appear". However compilers, obviously, would have no problem in distinguishing comments from source code.

Comments are not terminated by an end-of-line since the free format nature of the language does not recognize the end-of-line entity.

Requirement: H8

The language will not permit unmatched parenthesis of any kind.

---- T ----

Requirement: H9

There will be a uniform referent notation.

---- T ----

Entire variables are referenced by the variable identifier name. A component of a variable is referenced by the denotation of the variable identifier followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable. However, for each structuring type references are consistent and this evaluator assumes that this is the intent of this requirement.

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---- T ----

The grammar of PASCAL is truly context free.

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made irrevocable when the language is defined or explicitly in each program.

---- T ----

The structure of PASCAL requires all identifiers to be defined prior to reference and each executable statement is entirely explicit thus eliminating all of the traditional defaults. However, the collating sequence of the alfa character set is not a function of the language and will differ from one implementation to the other. This characteristic could be considered an unalterable default though it is unlikely to have anymore than a minor impact if the programming authorities are aware of the collating sequence for a specific implementation.

Requirement: I2

Default capabilities will be provided for special capabilities affecting only object representation on other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---- U ----

This requirement appears to be primarily addressing the internal algorithms of a compiler which affect the object representation of data and object code generated but do not affect the functional logic. The PASCAL language definition does not address this subject.

Requirement: I3

The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---- P- ----

Although the language includes the feature of associating data

constants with compile-time identifiers, as defined in section D1 of this report, however, those identifiers may only be used locally and do not address the object machine environment. The required features could be added to the language at modest cost, however, the cost of a given system would be closely related to the characteristics of that system.

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compiler time variables. In such cases, the conditional will be evaluated at compile-time and only the selected path will be compiled.

---- T/U ----

The PASCAL language contains the necessary features to satisfy this requirement (i.e., compile-time identifiers and case statements), however, it is not defined if compile-time evaluation of "constant" expressions and conditional compilation is a language feature. In general, this decision would be left to the compiler designer. It would, of course, be trivial to necessitate such features by including them in the language definition.

Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---- T ----

This language, like its predecessor ALGO-60 contains a minimum base language from which more complex components may be constructed. Declarative statements are of two (2) basic types; data declarations and procedure/function declarations. Executable statements are of eight (8) basic types; assignment, procedure invocation, if, case, while, repeat, for, and go to resulting in a total of ten (10) basic statements. Though not a theoretical minimum, the power realized by this choice of statements would appear to be close to optimum.

The design objectives of PASCAL included; systematic structure, flexibility of program and datat structuring, and efficient implementability. This reviewer feels that those goals have been met.

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language.

---- F ----

Information pertaining to translator dependent limits are not definable within the language. The effort to include a translator environment definition would, in most instances, be major.

Requirement: I7

Language restriction which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---- T ----

As with other requirements concerning translator and object environments, information was not available from the language definitions. The language itself does not contain any restrictions which are dependent upon the object environment, however, it is not unreasonable to assume that translators exist with such restrictions.

Requirement: J1

The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---- P ----

The source material used for this evaluation was limited to a language definition and did not address virtual or real translators. It is this reviewers opinion that the efficiency of the object representation is, to no small extent, a function of the ingenuity of the translator designer and the environment in which the object code will be invoked. However, the highly structured nature and syntactic consistency of the language will provide a hospitable environment for implementing efficient and reliable translators on presently available computers.

Requirement: J2

Any optimizations performed by a translator will not change the effect of the program.

---- T ----

As previously explained, information performing to specific translators was not available to this reviewer. It would, however, appear to be mandatory for any translator optimization that the program logic not be effected.

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---- P ----

PASCAL does not support machine language insertions, however, extending the standard procedure set would provide a mechanism for referencing encapsulated machine routines. Such routines would require a detailed knowledge of the parameter handling protocal and thus would be a function of the translator implementation and the run-time executive environment.

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will distinct form the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

---- F ----

The physical description of the object representation of composite data structures cannot be specified from within the source of PASCAL programs. It is, however, feasible that a translator could include optimization based upon the logical description and usage.

The anticipated effort of including explicit physical specifications of data objects would be major.

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---- F ----

The PASCAL language definition implies that all procedure definitions will be implemented as closed subroutines thus this requirement is not met. The inclusion of an open routine facility would be moderate if it were explicitly defined as such. However, if the translator were required to optimally determine an open or closed implementation the impact would be major.

PASCAL EVALUATION SUMMARY

This section provides a top level summary of the evaluation of the PASCAL language. Those functions of the language which generally satisfy the HOL specification are highlighted, as are the deficiencies. Deficiencies are further classified into those which would require major and those which would require minor modification to satisfy the HOL requirements.

The development of PASCAL was based on two principal aims. The first was to make available a systematic, disciplined programming language based on a minimum number of fundamental concepts which were clearly and naturally reflected by the language. The second was to develop implementations of the language which were both reliable and efficient on presently available computers, thus dispelling the commonly accepted notion that useful languages must be either slow to compile or slow to execute, and the belief that any non-trivial system is bound to contain mistakes forever. A further aim was to provide data structures and functions to make it possible to solve both commercial and scientific problems to help erase the mystical belief that those disciplines must be segregated.

PASCAL includes the following characteristics which satisfy the requirements of the HOL specification:

Concise, well defined syntax providing maximum structural flexibility with the minimum of primitives and syntactic structures. Few keywords are required; word abbreviations are not permitted and language statements are free format.

Strongly typed identifiers, including pointer types, with well defined semantics which generally prohibit implicit data type conversions; extensible data types by enumeration, sub-ranging and powersets.

Structured complex composite data structures, i.e., arrays of records, etc., providing almost unlimited compositions of the primitive data types.

Block structured environment providing clearly discernible scopes of activity and limited recursive capabilities.

Minimum set of pre-defined standard functions and procedures which may readily be extended to optimize system performance for specific purposes.

Universal usage of expressions, of appropriate type, wherever a value is required.

Minimization of side effects by limiting identifier references in functions to local identifiers and disallowing assignments to formal parameters.

With only one exception, defaults are not permitted.

Language structure is such that efficient implementations should be readily obtained.

Conversely, PASCAL includes the following characteristics which either conflict or are deficient.

Absence of fixed-point data types.

Absence of range and resolution for floating point data elements.

Alternate record structures are selected by a single "tag" field in conflict with the HOL requirement for selection by a general Boolean expression.

Relaxation of type checking for assignments when dependent variable is real and expression is integer or integer subrange and where expression is subrange of dependent variable type.

Variable number of arguments to procedures is not supported.

Absence of explicit control structures for parallel processing, exception handling and asynchronous interrupts.

Language does not address real-time control functions, i.e., control path delay, etc.

Apparent absence of conditional copilation facilities based upon compile time variables specifying object computer environment.

Physical description of object data representation is not supported.

Open subroutines are not supported.

Recursive subroutines are not explicitly defined but are only implied by loca references.

Modification of PASCAL to more fully support the HOL specification would be feasible. As detailed in this report, some of the modifications would be minor, while others would be relatively major efforts, and costly. Those in the former group include; fixed point data types, range and resolution specification, compile or load time evaluation of constant expressions, etc. Those in the latter group include: extension of operators to new data types, seperation of allocation and access scopes, parallel processing and selection of open/closed subroutines. In addition to these latter items, there are a number of requirements which are in conflict with the basis of PASCAL and could not readily be solved without changing the concept of that language. The most significant of these are those concerning the treating of subrange types as of the same general type as the base type and the concept of generic procedures where the formal parameters are not fully specified. It is believed by this reviewer that the intent of the former is satisfied by relaxing type checking between base and subrange types. The latter poses more of a problem since it is difficult to envisage how formal parameter specifications could be made optional without being inconsistent with the type checking convention of the language and in fact, the general HOL requirements of type consistency.

In conclusion, it is the belief of this reviewer that PASCAL satisfies most of the salient point of the HOL requirement. With modifications and the acceptance of meeting the intent of some HOL requirements though not necessarily explicitly PASCAL should encompass the total HOL requirements.

EVALUATION OF

CS-4

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Suite 16
Reston, Virginia 22090

December 3, 1976

This report presents an evaluation of the basic CS-4 language with the requirements listed in the "Tinman". (DOD requirements for high order computer programming languages, "Tinman" - 1 March, 1976, Section IV) For purposes of comparison CS-4 is considered to be defined by:

CS-4 Language Reference Manual and Operating System Interface.
Intermetrics,inc.,Cambridge,Mass.,October,1975.

There are 78 language requirements listed in section 4 of the "Tinman". This report compares basic CS-4 with each individual requirement. A summary of the degree to which the language satisifies each requirement is presented.

The introductory paragraph of each "Tinman" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meaning are as follow:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DoD "Tinman" requirements.

Note: In the following text references to "CS-4" always imply basic CS-4. Any references to full CS-4 will be explicit.

A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

---T---

CS-4 totally meets this requirement.

Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), boolean and character and will provide arrays (i.e., composite data structures with indexible components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

---P---

Integer.....T
Fixed point.....F
Floating point.....T
Character string.....T
Boolean.....T
Arrays.....T
Records.....T

CS-4 does not provide a seperate fixed-point mode. The CS-4 mode FRACTION provides for the special case of a floating real with an exponent of zero.

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---T---

CS-4 meets this requirement totally for all numeric data modes.

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

---F---

CS-4 does not support fixed point typed data.

Requirement: A5

Character sets will be treated as any other enumeration type.

---F---

CS-4 provides for a single character set (revised ASCII).

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

---T---

Number of dimensions fixed at compile time.....T
Type fixed at compile time.....T
Lower subscript bound fixed at compile time....T
Subscripts from contiguous range.....T
Subscripts from enumeration type.....T
Upper subscript bound fixed at scope entry.....T

CS-4 totally meets this requirement.

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

---T---

CS-4 totally meets this requirement.

B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

---T---

Variable declaration for all data types.....T
Encapsulated type declaration.....T
Array or record declaration.....T

CS-4 totally meets this requirement.

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---T---

CS-4 totally meets this requirement.

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---T---

CS-4 totally meets this requirement.

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

---P---

Addition.....T
Subtraction.....T
Multiplication.....T
Division (with real result).....T
Negation.....T
Exponentiation.....T
Division with remainder.....F

CS-4 provides the majority of the arithmetic operations specified in B4 with the following execeptions:

(a). The CS-4 operation IDIV provides for integer division of the various numeric types with no capability for access of remainder.

(b) Fixed point operations are not supported (see requirement A2).

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

---T---

CS-4 totally meets this requirement.

Requirement: B6

The built-in boolean operators will include "and", "or", "not", and "nor". The operations "and" and "or" will be evaluated in short circuit mode.

---T---

Short circuit "and".....T
Short circuit "or".....T
Not.....T
Nor.....T

CS-4 totally meets this requirement.

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---T---

Assignment between records and arrays.....T
Scalar operations on arrays.....T

CS-4 totally meets this requirement.

Requirement: B8

There will be no implicit type conversions but no conver-

sion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

---P---

Explicit conversions.....U
No implicit conversions.....T
Explicit between scale factors.....F

Implicit conversion between modes in CS-4 is not premitted. No explicit conversion operations are defined in the available documentation.

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

---T---

CS-4 totally meets this requirement.

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---F---

"The CS-4 Operating System Interface" provides extensive I/O operations, these facilities however are extensions of the basic language definition and therefore are not included in this evaluation. No I/O operations are defined for basic CS-4.

Requirement: B11

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

---T---

CS-4 totally meets this requirement.

C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

---T---

CS-4 totally meets this requirement.

Requirement: C2

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---P---

Unambiguous presentation.....T
Few precedence levels.....T
Require explicit parentheses.....F
No user defined levels.....T

Explicit parentheses are not required in expressions involving operators of equal precedence.
Full CS-4 provides facilities for defining new pre-,post-,and infix operators and attaching precedence to their evaluation. Existing operator precedence cannot be redefined.

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to

variables of that type are allowed.

---T---

CS-4 totally meets this requirement.

Requirement: C4

Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

---T---

CS-4 totally meets this requirement.

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

---T---

Consistency in parameter rules.....T
No special parameter operations.....T

CS-4 totally meets this requirement.

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

---T---

Dimensions determined at compile time.....T
Size and subscripts can be passed.....T
Type agreement for actual and formal parameters.....T

CS-4 totally meets this requirement.

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P---

Parameter constants.....T
Parameter variables.....T
Exception conditions.....F
Procedure parameters.....T

CS-4 does not provide a formal parameter class for exceptional condition control.

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

---T---

Optional properties.....T
Fixed at run time.....T

CS-4 totally meets this requirement.

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

---F---

Variable number of arguments.....F
All but constant number same type.....F
Number fixed at compile time.....F

CS-4 does not support procedures with a variable number of arguments. Full CS-4 does provide the necessary mechanisms for this requirement.

D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

---T---

CS-4 totally meets this requirement.

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---T---

Literals of built-in data types.....T
Consistency in value.....T

CS-4 totally meets this requirement.

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---T---

Declare initial values.....T
Allocation scope initialization.....T
No default values.....T

CS-4 totally meets this requirement.

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---P---

Range specification is required for all numeric modes (specifictaion may be implicit).
Fixed point typed data is not supported (see note on re-

quirement A2).

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

---T---

CS-4 totally meets this requirement.

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

---T---

Shared/recursive substructure capability.....T
Variable/record/array componenT handling.....T
Typed pointer characteristic.....T
Allocation never wider than access.....T

The CS-4 pointer mechism is limited to dynamicly allocated data (data allocated in HEAP storage via the ALLOCATE statement).

E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

---T---

CS-4 totally meets this requirement.

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

---T---

CS-4 totally meets this requirement.

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---T---

CS-4 totally meets this requirement.

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---T---

CS-4 totally meets this requirement.

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

---T---

CS-4 totally meets this requirement.

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of

disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

---T---

Enumeration.....T
Cartesian products.....T
Discriminated union.....T
Power set of enumeration type....T

CS-4 contains extensive data type definition facilities. Enumeration and power set definition can be applied through the use of the RANGE qualifier.

Requirement: E7

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

---T---

Does not permit free unions.....T
Does not permit subsetting.....T

CS-4 totally meets this requirement.

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

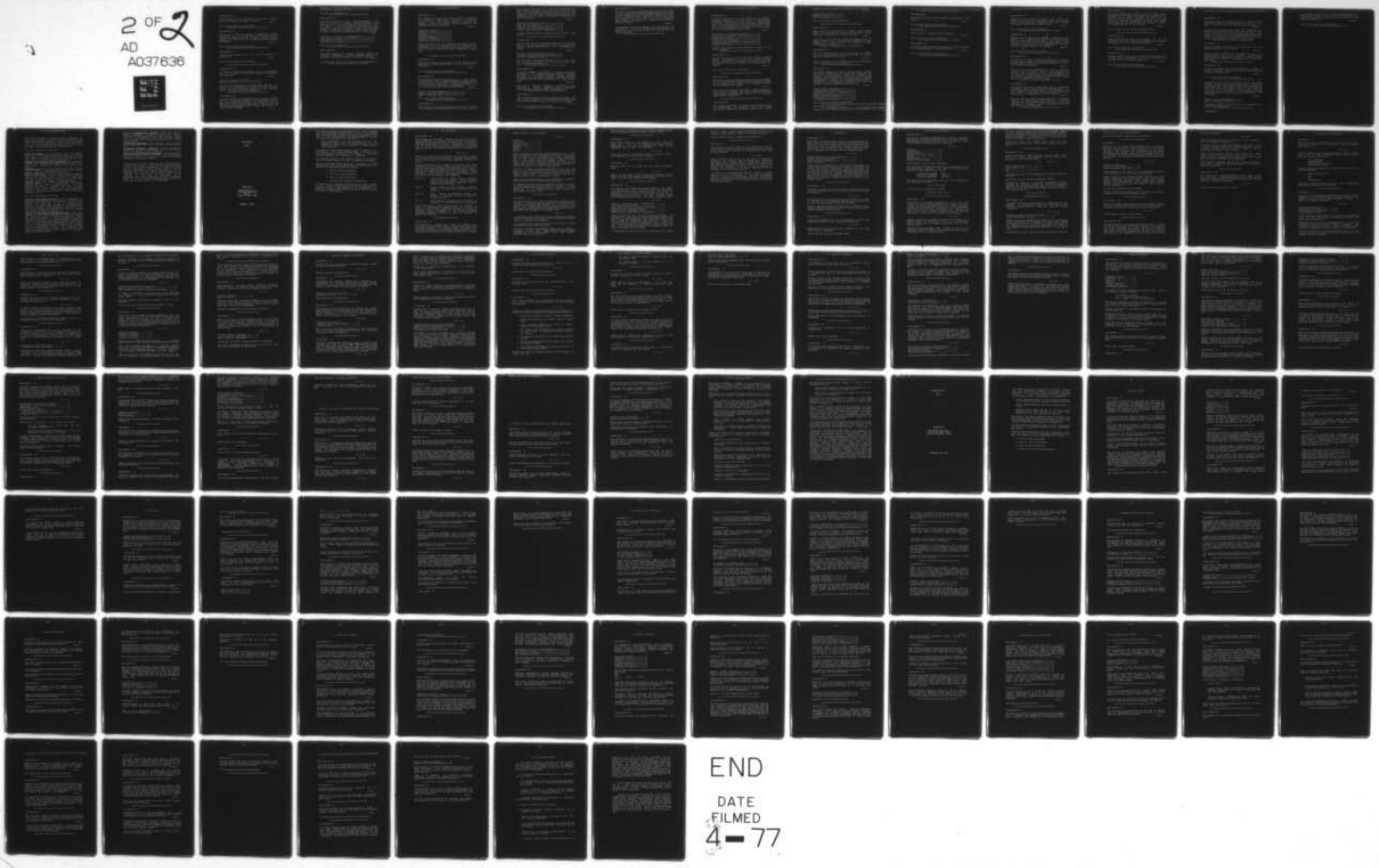
---T---

CS-4 totally meets this requirement.

AD-A037 636 RLG ASSOCIATES INC RESTON VA F/G 9/2
EVALUATION OF CORAL 66, PASCAL, CS-4, TACPOL, CMS-2.(U)
NOV 76

UNCLASSIFIED NL

2 OF 2
AD
A037636



END

DATE
FILMED
4-77

F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---T---

CS-4 totally meets this requirement.

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

---T---

CS-4 totally meets this requirement.

Requirement: F3

The scope of identifiers will be wholly determined at compile time.

---T---

CS-4 totally meets this requirement.

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accesible in the language.

---U---

Variety of data and operations.....U
Easily accesible.....U

The compostion and extent of application oriented libraries is not defined for CS-4 at this time. The creation of such libraries should present no great problems at the time CS-4 is finally implemented.

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---U---

Accesible at compile time.....U
Other source language routines.....U

See note on requirement F4.

Requirement: F6

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---U---

Libraries and compools indistinguishable.....U
Hold anything definable in language.....U
Many levels of access.....U
Many specialized subsets.....U

See note on requirement F4.

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---T---

The MSTRUCTURE mode of CS-4 satisfies this requirement.

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

---P---

Sequential control.....T
Conditional control.....T
Iteration.....T
Recursion.....F
Parallel processing.....F
Exception handling.....T
Asyncronous interrupts.....T

CS-4 provides all of the standard sturctured control mechanisms with the exception of parallel processing and recursion. Full CS-4 provides for the inclusion of control structures for both parallel processing and recursion.

Requirement: G2

The source language will provide a "go to" operation applicable to program labels within its most local scope of definition.

---T--

CS-4 totally meets this requirement.

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---T---

Based on boolean expression.....T
Based on type from discriminated union.....T
Based on computed choice.....T

CS-4 totally meets this requirement.

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will re-

quire control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g. fixed number of iteration or elements of an array exhausted).

---P---

Termination anywhere in loop.....T
Local control variables.....F
Entry at loop head only.....T
Efficient and clear simple cases.....T
Control value available at termination.....T

In CS-4 iterative control variables are not treated specially.

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---F---

Recursive and nonrecursive routines.....F
Explicitly specify recursive routines.....F
No nested recursive procedures.....F

CS-4 does not provide for recursion in any form. Full CS-4 supplies facilities for recursive routines which meet or exceed requirement G5.

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---F---

CS-4 has no parallel processing facilities. full CS-4,however, provides extensive parallel processing mechanisns which meet or exceed requirement G6.

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

---T---

CS-4 totally meets this requirement.

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

---F---

The necessary features to satisfy this requirement are contained in "the CS-4 Operating System Interface" or within full CS-4. Basic CS-4 does not include the required mechanisms.

H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

---T---

Free format with statement separator.....T
Mnemonically significant identifiers.....T
Conventional forms.....T
Simple uniform grammar.....T
No special case notations.....T
No abbreviation of keywords or identifiers....T
Unambiguous grammar.....T

Although CS-4 syntax is quite extensive, it seems to be highly readable.

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedence.

---T---

CS-4 totally meets this requirement.

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

---P---

CS-4 uses the 94 printing characters of 128 character revised USACII character set (USA standard X3.4-1968). modification of the language to accept the 64 character subset should be trivial.

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break

character for use internal to identifiers and literals.

---T---

Literals for numbers.....T
Character strings.....T
Break character.....T

CS-4 totally meets this requirement.

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---P---

CS-4 lexical units may not be continued across lines. at present there is no provision for inclusion of object characters in literal strings. The addition of this feature should be trivial.

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

---T---

given the extent of the language, keywords are acceptably few in number and all are reserved.

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

---P---

Single comment convention.....F
Distinguishable from code.....T
Introduced by one or two characters.....T
Contain any charcter.....P
Appear anywhere reasonable.....T
Terminate at end of line.....P
Not prohibit reformatting.....T

CS-4 contains two comment conventions:

(1) The precent character (%) may be used to indicate a commen terminates on end of line.

(2) The bracket notation ({ ,}) form requires both comment delimiters to be present. termination is not automatic on end of line.

also each comment delimiter within the comment string must be matched.

Requirement:H8

The language will not permit unmatched parentheses of any kind.

---T---

CS-4 totally meets this requirement.

Requirement: H9

There will be a uniform referent notation.

---T---

CS-4 totally meets this requirement.

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---T---

CS-4 totally meets this requirement.

I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic.That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

---T---

CS-4 totally meets this requirement.

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---T---

CS-4 totally meets this requirement.

Requirement: I3

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---P---

CS-4 does not include provisions for a special class of "compile time" variables. Instead any variable who's value is known at compile time (via initialization) may be utilized in compile time functions. Specification of the object machine configuration is accomplished through the use of %STRUCTURE specifications.

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

---P---

CS-4 provides for compile time evaluation of expressions composed of known values. There are,however, no mechanisms whereby selective compilation may occur. Full CS-4 supplies the necessary compile time control structures for meeting this requirement.

Requirement: I5

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---T---

CS-4 is based upon the kernel language ELS-K.

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

---T---

CS-4 totally meets this requirement.

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---T---

CS-4 totally meets this requirement.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---U---

The ability or inability of a given CS-4 translator to produce efficient object code is an open question at this time. Certainly the CS-4 language designers desire efficiency of object code. Such a property,however,cannot be defined in the design of a language directed at a number of computers possesing a variety of architectures.

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

---T---

CS-4 meets this requirement in the sense that it is a stated design goal of the language. In reality the determination of this question would be somewhat dependant upon the specific implementation of a given CS-4 translator.

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---T---

CS-4 totally meets this requirement.

Requirement: J4 It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapslated and will distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

---p---

Specify object presentation.....T
Specify time/space tradeoff.....U

Time/space tradeoff specification is not addressed in the available documentation.

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---T---

CS-4 totally meets this requirement.

CS-4 EVALUATION SUMMARY

This section presents a summary of the evaluation of basic CS-4 against the DOD "Tinman" requirements. The major strengths and weaknesses of the language in relation to the requirements will be detailed and appraisal given as to the overall desireability of utilizing CS-4 as a base language for future HOL developement.

The major strengths of CS-4 are:

Strong data typing. All data modes in CS-4 are rigidly typed as to data characteristics and access. In addition type checking is enforced during runtime as well as compile time. such a trait significantly increases the possiabilities of producing reliable programs.

Extensive and logical control structures. CS-4 provides a concise set of control mechanisms sufficient for structured program construction without being overly elaborate.

Extensiability. CS-4 contains features for the definition of new data modes and for the utilization of these extended modes in a straightforward manner.

Machine independance. CS-4 is a highly machine independant language with a set of facilities for mating the language to a specific machine. Whether the facilities provided are sufficient for a tight fit is an open question that can only be resolved by implementation.

Levels of usage. CS-4 provides facilities whereby the language may be used on several levels. This multileveling includes the enabling/disabling of GOTO usage,run-time range checking and other language features. This type of usage scheme would seen well suited to the stepped developement of both programs and prorammer skill.

The major weaknesses of CS-4 are:

Lack of a fixed-point data mode. The CS-4 designers rationalized the exclusion of fixed-point as being both unnecessary due to the drop in cost of floating point hardware and inefficient because of the necessity of shifting for decimal point alinement in arithmetic operations. The validity of these arguments would seen to hold for medium to large scale systems while becoming suspect when applied to most mini systems.

Lack of I/O facilities in the base language. Basic CS-4 contains no I/O facility definitions. Instead I/O mechanisms are defined within the context of "The CS-4 Operating System Interface", a set of procedure calls which define,in addition to I/O services,parallel processing constructs and other miscelaneous operating system functions. Embedded within this "interface" is the definition of a complete file system structure. The inclusion of such a specification as part of the basic language would be unacceptable as being too extensive (the language designers have in effect defined an operating system for their language).

CS-4 is an unimplemented language. This might not be fairly considered a defect in relation to "Tinman" requirements,however in relation to the overall consistancy of the language it would seen that much benifit could be gained by the feedback resulting from a trail implementation.

No recursive procedures. CS-4 procedures have no capability for recursion. This defect has been corrected in full CS-4.

No parallel processing capability. Parallel processing constructs ,like the I/O mechanisms of CS-4, are embedded within the "Operating System Interface".

Lack of concise conversion procedures. CS-4 documentation is quite specific that no implicit conversion between unlike data modes will occur. No explicit conversion mechanisms are specified however.

In summary CS-4 is an extensive,near state-of-the-art language which contains a great number of the traits outlined in the "Tinman" requirements. The extended version of the language,full CS-4,contains nearly all of the necessary features for meeting the requirements. On the negative side CS-4 has several serious defects,all of which seem to be outgrowths of the fluid nature of the language. If this lack of consistancy can be corrected,either through feedback from trail implementations or a more unified language design effort, CS-4 would make a fine choice for a HOL baseline.

EVALUATION OF

TACPOL

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Suite 16
Reston, Virginia 22090

December 3, 1976

This report presents an evaluation of the TACPOL language with the requirements listed in the "TINMAN". (DOD Requirements for High Order Computer Programming Languages, "TINMAN" - 1 March 1976, Section IV). For purposes of comparison TACPOL is considered to be defined by:

> CPCEI SPECIFICATION FOR COMPILER/ASSEMBLER FOR FIRE DIRECTION SYSTEM, Spec. No. EL-CG-00043082C, Doc. No. 595909-600C, Volume 1 of 2, Appendix 10 (FORMAL DEFINITION OF TACPOL)

There are 78 language requirements listed in section 4 of the "TINMAN". This report compares TACPOL with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The paragraph number of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meaning are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A summary of the evaluation is included as the last section of this report. Merits and failures of the language as it currently is defined are discussed. Also discussed are the potential language modifications that can be made to support DOD "TINMAN" requirements.

A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile-time and unalterable at run-time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

--- T ---

TACPOL is a strongly typed language. All quantities (units of storage which hold a value) must be one of four possible types: short numeric; long numeric; character string; bit string.

Quantity types cannot be altered at run-time but value conversions can be made by means of several "intrinsic procedures" (explicit conversion functions which may be used in the body of a TACPOL program and which have meaning independent of the definitions given by the programmer). These procedures (listed below) make appropriate conversions on the value of the argument of the procedure.

SHORT (X)	long numeric to short numeric, character string to short numeric, or bit string to short numeric (specific conversion selected depends on compile-time of X)
LONG (X)	short numeric to long numeric, character string to long numeric, or bit string to long numeric
CHAR (X)	short numeric to character string, long numeric to character string, bit string to character string
BIT (X)	short numeric to bit string, long numeric to bit string, or character string to bit string

TACPOL does support a cell structure which is designed to permit an equivalence of storage usage (i.e., overlay) at run-time. This is used principally to buffer I/O records of different formats. Depending on the run-time checking available (undefined in TACPOL), it might be possible to subvert type checking.

Requirement: A2

The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexible components of homogeneous type) and records (i.e., composite data structures with labeled components of hetero-

geneous type) as type generators.

--- P ---

Integer . . . . . P
Fixed Point . . . . . P
Floating Point . . . . . T
Character string . . . . T
Boolean . . . . . T
Arrays . . . . . T
Records . . . . . P

TACPOL supports the four data types described in A1. In the case of numerics, TACPOL includes the ability to state precision (number of bits representing the most significant part of the value) and scale (where to place the binary point). Short numeric permits a precision of from 1 to 31 bits; long numeric from 32 to 62 bits. The scale must be between -127 and +127. Operations on numerics take into account the precision and scale of each operand.

Although all numbers are stored internally in one word (32 bits) or two word (64 bit) quantities with the appropriate scale factors, the programmer may code number values as integers, fixed point, or floating point and they will be treated properly both on input and on output.

TACPOL also provides arrays (up to 3 dimensions) as quantity structures which may be declared and designated.

The concept of record is well understood in TACPOL, although a strong distinction is made between an externally recorded record (without much detail) and the well defined or typed quantities in main memory from/to which records are written/read (normally cells).

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

--- P ---

As discussed in A2, TACPOL does support precision specification on individual quantities but not any sort of global scoping of precision requirements.

All operations must take into account the individual precision specifications of each operand.

In terms of hardware independence, TACPOL does not specify a particular machine, but it seems strongly oriented toward machines with 8-bit bytes, 32-bit words, whose internal character set is 8-level ASCII.

Global scoping of precision could be added to TACPOL without much difficulty, through the addition of new syntax.

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile-time. Scale factor management will be done by the compiler.

--- T ---

Within the limits of available numeric precision, TACPOL fully supports the scaled integer feature.

Requirement: A5

Character sets will be treated as any other enumeration type.

--- F ---

TACPOL does not support the definition and ordering of characters by the source program. Characters are said to be in ASCII, one 8-bit byte equals ASCII character.

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile-time. The upper subscript bound will be determinable at entry to the array allocation scope.

--- P ---

Number of dimensions fixed at compile-time . . . . T
Type fixed at compile-time . . . . . T
Lower subscript bound fixed at compile-time . . . T
Subscripts from contiguous range . . . . . T
Subscripts from enumeration type . . . . . T
Upper subscript bound fixed at scope entry . . . . F

TACPOL supports one, two, and three dimensional arrays. The dimension of each array and the range of each subscript is fixed at compile (declaration) time. Subscript ranges must fall between one (always the lower bound) and N, where N is a number explicitly given at declaration time. N may be any type of numeric quantity which yields a value not less than one. A real quantity is truncated to the largest whole integer not greater than its value.

At designation, subscripts may be an expression which yields

a short numeric value within the declared range for that subscript; real values are reduced to integers as above.

Variable upper bounds on arrays are not supported.

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile-time. The name and type of each record component will be specified by the user at compile-time.

--- P ---

TACPOL places no restrictions on the use of alternative structures for records. Each file operation in TACPOL has associated with it a fully defined buffer data structure, with each element (scalar, array, table, group, cell) in the structure well defined and typed. There is nothing to prevent the programmer from reading one record into one structure and the next into another.

In addition, the programmer can declare a "cell" (a collection of sets of quantities) all of which are allocated storage in common (i.e., overlayed). Each set of quantities defines a potentially different data structure for a record. Even in this case, full type checking is done as the subcell quantities are used.

B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

--- P ---

Variable declaration for all data types . . . T
Encapsulated type declaration . . . . . T
Array or record declaration . . . . . T
Array assignment . . . . . P

TACPOL supports referencing of quantities and assignment of values to any type of quantity or structure which can be declared. In the case of assignment, multiple quantities can be assigned a single value in one assignment operation, but multiple quantities (e.g., arrays) cannot be assigned multiple values in a single operation. The language fully supports the ability to reference any low level element in a complex data structure. Reference always retrieves the last assigned value.

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

--- T ---

TACPOL supports the = relational operator which can be used to compare any two objects of the same type for identity. All four TACPOL-recognized types are supported.

Numeric values are compared algebraically, character string values character-by-character, left to right, and bit string values bit-by-bit, left to right.

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

--- P ---

TACPOL supports all six relational operators on all four types which it recognizes.

TACPOL does not recognize unordered sets.

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

--- P ---

Addition . . . . . T
Subtraction . . . . . T
Multiplication . . . . . T
Division (with real result) . . T
Negation . . . . . T
Exponentiation . . . . . T
Division with remainder . . . . P

TACPOL supports all of these operations.

All except for division with remainder are available in single syntactical operators. In order to do division with remainder, three operators are needed:

normal real division (A/B)
intrinsic procedure TRUNC (C)
intrinsic procedure REM (D)

The first two can be confined to get the whole dividend of the division as follows:

DIV = TRUNC (A/B)

The remainder is obtained as follows:

REMAINDER = REM (A,B)

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

--- P ---

Rounding takes place in TACPOL only when the intrinsic procedure, ROUND, is explicitly called by the programmer. Round operates only on the least significant part of the numeric value.

Truncation can be explicitly done in TACPOL by means of the intrinsic procedure TRUNC, which operates only on the least significant part of a numeric value.

Implicit truncation takes place during an assignment operation when the defined precision of quantity being assigned a value is less than that of the value derived from the assignment expression. Once again, only the least significant part of the numeric value is affected.

Rounding of floating point numbers takes place only when specifically called for. The implicit default is truncation.

Requirement: B6

The built-in Boolean operators will include "AND", "OR", "NOT", and "NOR". The operations "AND" and "OR" will be evaluated in short circuit mode.

--- P ---

Short circuit "and" . . . . . U
Short circuit "or" . . . . . U
Not . . . . . T
Nor . . . . . P

Short-circuit mode is not addressed in the TACPOL FORMAL DEFINITION.

Normal AND, OR, and NOT are supported by TACPOL.

NOR does not exist as an explicit syntactical element, although the identical effect of NOR can be achieved by means of the intrinsic procedure BOOL as follows (where X and Y are the bit variables being NORed, and the bit literal is the truth table):

BOOL (X,Y, '1000'B)

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

--- P ---

Assignment between records and arrays . . . . P
Scalar operations on arrays . . . . . F

TACPOL supports the assignment of values of one set of quantities to a second set of quantities by means of the intrinsic procedure, MOVE (X,Y). There is no requirement that the two structures (e.g., arrays) be conformable; the length of the shorter of the two determines the scope of the operation.

The FORMAL DEFINITION does not define the scope or nature of

type checking in the MOVE procedure.

Scalar operations on arrays are not supported.

Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

--- P ---

Explicit conversions . . . . . P
No implicit conversions . . . . . T
Explicit between scale factors . . T

TACPOL supports a full range of type conversions (see answer to Requirement A1), for the types it recognizes.

Numeric values are all stored internally as one or two word binary quantities with a scale factor, hence numeric operations (within short or within long) always yield numeric values. The requirement for explicit conversion operations between integer, fixed point, and floating point internal data is undefined.

No implicit conversions are supported.

Numeric values can be rescaled (with no change in precision) by means of the intrinsic procedure, SCALE.

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run-time exception condition when any integer or fixed point value is truncated.

--- F ---

TACPOL does not support this function.

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be in-

stallation dependent.

--- p ---

TACPOL provides support for a variety of file declarations and file processing operations.

In TACPOL, files are named and declared to be either serial or direct (by key), partitioned or nonpartitioned, with concurrent or nonconcurrent I/O; a file is opened in one of three modes -- input, output, update.

Defined file operations include: OPEN, CLOSE, READ, WRITE, WRITE ENDFILE, REWRITE, DELETE, SPACE, REWIND, UNWIND. Exception condition processing on end-of-file, invalid record key, and invalid partition key is supported.

TACPOL does not consider devices or channels explicitly, only files. Assignment and reassignment of I/O devices is therefore undefined.

Requirement: B11 .

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

--- F ---

Power sets are not defined in TACPOL.

C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

--- P ---

Numeric expression evaluation within TACPOL is done according to the following precedence order of operators (from highest to lowest precedence):

infix operators
exponentiation
multiplication/division
addition/subtraction

Boolean expression evaluation is done according to the following precedence:

relational operators
NOT
AND
OR
concatenation

Within any precedence level and for all other expressions, evaluation is performed left to right.

Requirement: C2

Which parts of an expression constitute the operands to each operation, within that expression, should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

--- P ---

Unambiguous presentation . . . . . P
Few precedence levels . . . . . T
Require explicit parentheses . . . . P
No user defined levels . . . . . T

TACPOL expression presentation is relatively straightforward, with the usual order of operator precedence. (See answer to Requirement C1.)

Parenthesis are permitted to reduce ambiguity or to alter precedence, but are not required except in the case of a relational subexpression within a Boolean expression, where the entire subexpression must be enclosed in parentheses.

Two unusual features of expression notation which (in this reader's opinion) detract from overall readability of expression are the operators:

(**) and (*)

which are forms of exponentiation and multiplication, but which create long numeric result values even though the operands are of the short numeric type.

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

--- P ---

TACPOL meets this requirement in most cases, but there are some minor exceptions. For example, in the designation of a substring, the character count argument can only be an integer literal, not an expression.

Requirement: C4

Constant expressions will be allowed in programs where constants are allowed, and constant expressions will be evaluated before run-time.

--- P ---

In TACPOL, constant expressions can replace constants anywhere in the program, except in type specifications, array declarations, table declarations, value declarations, and in certain arguments of intrinsic procedures. These exceptions are not regarded as major shortcomings.

The evaluation of constant expressions before run-time is generally undefined in the FORMAL DEFINITION.

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

--- P ---

Consistency in parameter rules . . . . . P
No special parameter operations . . . . P

Generally, the usage of parameters appears similar. This is especially true in simple cases. However, as the user takes advantage of the more sophisticated data structures in TACPOL (e.g., cells), the rules for parameter usage start to

vary, depending on what is needed to describe the structure.

Parallel processing is not defined. Exception handling is defined only for file I/O operations and has a syntax all its own.

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile-time. The size and subscript range for array parameters need not be determinable at compile-time, but can be passed as part of the parameter.

--- p ---

Dimensions determined at compile-time . . . . . T
Size and subscripts can be passed . . . . . F
Type agreement for actual and formal parameters . . . . T

In TACPOL, the dimensions of an array are fully determined at compile-time. This includes the number, size, and subscript range.

Size and subscript range cannot be passed as part of the parameter.

Formal and actual arguments must always agree in type.

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

--- p ---

Parameter constants . . . . . T
Parameter variables . . . . . T
Exception conditions . . . . . P
Procedure parameters . . . . . T

TACPOL does not classify formal parameters in a universal sense although it does meet some of the requirements stated.

When formal parameters are listed in a procedure declaration, they must also be defined by an imbedded parameter declaration which classifies each as a quantity (variable), value (constant), procedure parameter, or point (label).

Exceptions, per se, are defined only for file I/O, and permit the execution of a single statement in the event that

one of the three conditions -- end-of-file, invalid record key, invalid partition key -- are met. The structure for this is:

ON condition THEN statement ELSE statement

Other run-time conditions which can be checked are divide by zero, fixed overflow, or reference to a specified identifier name. The desire to check for such conditions is declared by the programmer. Upon occurrence, a "snap procedure" (mentioned but undefined in TACPOL) is to be implicitly invoked. The structure of an interface to such a procedure is not discussed in the FORMAL DEFINITION.

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run-time.

--- P ---

Optional properties . . . . . F
Fixed at run-time . . . . . T

TACPOL requires that all formal parameters in a declared procedure be themselves declared by type, precision, dimension, and scale. Range is implicit in short and long scalar definition. Format is not addressed.

Procedure parameters are not alterable at run-time.

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile-time.

--- P ---

Variable number of arguments . . . . . F
All but constant number same type . . F
Number fixed at compile-time . . . . . T

TACPOL does not support variable numbers of arguments.

The number of required arguments for a procedure is fixed and fully determinable at compile-time.

D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

--- T ---

TACPOL provides this capability.

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

--- T ---

Literals of built-in data types . . . T
Consistency in value . . . . . T

TACPOL supports this capability.

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

--- T ---

Declare initial values . . . . . T
Allocation scope initialization . . . . T
No default values . . . . . T

Using the value declaration statements, the user can declare the initial value of individual variables. If he does not, no default values are assigned.

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

--- P ---

TACPOL requires that all numeric type variables be declared with a range which is composed of a precision (mandatory) and a scaling (optional, default is zero). The interpretation of range is for both compile-time and object time values (no difference is discussed).

TACPOL does not specifically address fixed point variables or their step size.

Within range specification, a precision of 1 to 31 bits implies short numeric type, and one of 32 to 62 bits implies long numeric type.

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

--- P ---

TACPOL supports this feature, except that user-defined types and ennumeration types are not defined.

Requirement: D6

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

--- F ---

Shared/recursive substructure capability . . . . F
Variable/record/array component handling . . . . F
Typed pointer characteristic . . . . . F
Allocation never wider than access . . . . . F

The concept of a pointer mechanism (i.e., indirect addressing) is not defined in the FORMAL DEFINITION, except in one location (section 10.6.8, Proper Procedure Designations) where the metalinguistic element <ADDR DESIG> (one argument designation) is defined to be /<BIT EXPR>/. Nothing else is mentioned anywhere else. The implication is that a hard address can be physically built using Boolean operations. There appears to be no other mechanism in the language to deal with pointers.

E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

--- F ---

TACPOL does not support these features.

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

--- F ---

Defined types are not supported in TACPOL.

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

--- P ---

TACPOL requires that many program components be defined explicitly. There are certain exceptions, however, such as:

- Scale factor defaults to zero on numeric quantities.
- Precision defaults to 31 on short numeric quantities.
- Data alignment defaults to PACKED or ALIGNED, depending on declaration type.
- For literals, E00 is default, and scaling defaults to 3.322 times the number of fractional digits in the literal.
- In substring operations, the optional count defaults to one.
- Several file-processing syntax elements can default to implicit values.
- All intrinsic procedures have default values and implicit range restrictions.

In addition, there are numerous implicit limits enforced by TACPOL, such as:

- The precision of numeric values.
- The maximum length of character strings (512) and bit strings (32).
- The maximum value of the step variable in a DO statement (32,767).

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

--- F ---

TACPOL does not support the definition of new types, and hence not the extension of existing operators of these types.

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

--- F ---

TACPOL does not support any of these features.

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile-time.

--- F ---

TACPOL does not support the capability of having user-defined types. Only built-in types are allowed.

Requirement: E7

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

--- T ---

Does not permit free unions . . . . T
Does not permit subsetting . . . . . T

TACPOL meets this requirement since no method of type definition is permitted.

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

--- F ---

TACPOL does not support user-defined types.

F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

--- P ---

TACPOL explicitly addresses scope of allocation by means of a comprehensive set of data structures which can be declared.

Scope of access is not addressed explicitly, although it can be derived from data structure designations in the program. Explicit block structures with local scoping of variables are supported.

The cell structure in TACPOL permits several different substructures to be allocated the same physical storage.

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

--- P ---

TACPOL does not support user-defined data types, nor does it distinguish between normal and "special" structures.

TACPOL does support block structures, with local scoping of variables.

Requirement: F3

The scope of identifiers will be wholly determined at compile-time.

--- T ---

TACPOL meets this requirement.

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

--- P ---

Variety of data and operations . . . . P
Easily accessible . . . . . P

TACPOL supports the program use of COMPOOL data (although the FORMAL DEFINITION does not clearly state how). In addition, a variety of intrinsic procedures are available, although this set is predefined and not alterable by the user.

By means of the CALL statement, separately created external procedures can be accessed. Potentially, this feature could be used to access user libraries. This feature is not described in any detail in the FORMAL DEFINITION.

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile-time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

--- P ---

Accessible at compile-time . . . . . T
Other source language routines . . . P

Although the FORMAL DEFINITION does not describe exactly how, TACPOL can support separate program components which are CALLed as needed. If these components have been compiled by a TACPOL compiler using the CODE language feature, then they may be written in other source languages.

TACPOL does not, however, allow direct calling of external routines written in another source language without the CODE feature.

Requirement: F6

Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

--- U ---

Libraries and COMPOOLS indistinguishable . . . U
Hold anything definable in language . . . . . U
Many levels of access . . . . . U
Many specialized subsets . . . . . U

Libraries are not explicitly addressed in the FORMAL DEFINI-

TION, although the ability to call an external procedure implies their existence. COMPOOLS are said to exist, but no further definition of COMPOOL or of language mechanisms to address them is given. Hence, the required features are undefined.

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

--- P ---

TACPOL provides machine independent interface with peripheral equipment by means of its file processing features (READ/WRITE level). The "device" is designated only through some undefined external correspondence with the program file names. Some extension of this capability could provide an interface with other special hardware.

G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

--- P ---

Sequential control . . . . . T
Conditional control . . . . T
Iteration . . . . . T
Recursion . . . . . F
Parallel processing . . . . P
Exception handling . . . . . P
Asynchronous interrupts . . U

In addition to normal sequential execution, TACPOL supports the following control elements:

DO (WHILE) - iterative
GOTO - alter execution sequence
IF . . . THEN . . . ELSE - conditional

Recursion is undefined in TACPOL, and appears to be impossible since procedure bodies are physically inserted in-line (macro fasnion) during compilation.

Parallel processing is not defined except for the ability to do concurrent I/O.

Exception handling is defined only for three cases of I/O processing - end-of-file, bad record key, bad partition key. In addition, condition checking can be performed on divide-by-zero, fixed overflow, and variable usage.

Interrupts are not defined explicitly in TACPOL. The WAIT operation causes processing to be suspended until all I/O operations on a file have ceased.

Requirement: G2

The source language will provide a "GOTO" operation applicable to program labels within its most local scope of definition.

--- T ---

TACPOL meets this requirement.

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

--- P ---

Fully partitioned . . . . . F
Based on Boolean expression . . . . . T
Based on type from discriminated union . . F
Based on computed choice . . . . . T

The ELSE clause in the IF statement is optional.

The conditional expression in an IF statement must be a Boolean expression. The selection argument in the SWITCH intrinsic procedure (analogous to a CASE statement) must be a short numeric argument.

Discriminated unions are not defined in TACPOL.

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity, or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

--- P ---

Termination anywhere in loop . . . . . F
Local control variables . . . . . U
Entry at loop head only . . . . . T
Efficient and clear simple cases . . . . . T
Control value available at termination . . U

Execution termination of a DO, either as a result of control variable value or of WHILE state, takes place at the top of the loop, since these conditions are only evaluated at that time.

Control variable must be a short numeric value, but is otherwise undefined in the FORMAL DEFINITION as being local or global. Whether the end value of the control variable is available is not defined in the FORMAL DEFINITION.

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

--- P ---

Recursive and nonrecursive routines . . . . . P
Explicitly specify recursive routines . . . . . F
No nested recursive procedures . . . . . T

Recursive procedures are not defined in TACPOL, and appear to be impossible since procedure bodies are physically inserted in-line (macro fashion) during compilation.

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

--- F ---

TACPOL does not support parallel processing.

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

--- P ---

TACPOL supports exception handling only for three specific I/O cases -- end-of-file, bad record key, bad partition key. In addition, the following program conditions can be declared and (presumably) checked -- divide by zero, fixed overflow, variable usage. Exactly how these later conditions are processed, once detected, is not defined.

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, and which permit asynchronous hardware interrupts to be treated as any other exception situation.

--- F ---

These features are not supported by TACPOL.

H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple, uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

--- P ---

Free format with statement separator . . . . . P
Mnemonically significant identifiers . . . . . T
Conventional forms . . . . . P
Simple uniform grammar . . . . . P
No special case notations . . . . . P
No abbreviation of keywords or identifiers . . . T
Unambiguous grammar . . . . . T

TACPOL is basically of free format with an explicit separator (semicolon).

There are a few exceptions to this, however, including:

- The last statement in a CODE block must have ENDspace in columns 10-13
- Within certain control structures (e.g., DO, IF) the semicolon is not present in all cases.

It allows mnemonically significant identifiers, and utilizes generally conventional forms, except for some designations of complex data structures. At the top level, the grammar is uniform, but data structure designations can be far from simple or uniform.

Abbreviations of keywords are not permitted. The grammar seems to be unambiguous.

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new infix operator precedence.

--- T ---

TACPOL meets these requirements.

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

--- T ---

TACPOL uses a 50 character subset or the 64 character ASCII set.

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

--- T ---

Literals for numbers . . . . . T
Character strings . . . . . T
Break character . . . . . T

TACPOL meets these requirements. The break character is the underscore.

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

--- U ---

Neither of these requirements is defined in the TACPOL FORMAL DEFINITION.

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

--- T ---

TACPOL recognizes 97 key words, all of which are fairly informative in context.

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code,

will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

--- P ---

Single comment convention . . . . . T
Distinguishable from code . . . . . T
Introduced by one or two characters . . . . T
Contain any character . . . . . P
Appear anywhere reasonable . . . . . P
Terminate at end of line . . . . . F
Not prohibit reformatting . . . . . T

TACPOL comments begin with /* and end with */. They may contain any TACPOL character except / and *.

In TACPOL, comments may appear anywhere an arbitrary space may appear. Generally this is reasonable, although overzealous use of comments in certain contexts (e.g., between the keyword IF and the conditional expression which follows it) may make the program less, rather than more, readable.

There is no requirement in TACPOL that comments terminate at the end of a line. Only */ can terminate a comment.

Requirement: H8

The language will not permit unmatched parentheses of any kind.

--- T ---

TACPOL meets this requirement.

Requirement: H9

There will be a uniform referent notation.

--- P ---

In general, TACPOL uses the same notation for both function calls and identifier designations, i.e, the name of the function or the identifier followed by arguments in parentheses. However, the syntax of what appears in parentheses is highly dependent on the item (function or variable) being described.

Requirement: H10

No language defined symbols appearing in the same context

will have essentially different meanings.

--- p ---

Generally, TACPOL meets this requirement except for one case: the symbol = is used for both assignment and equality.

I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

--- p ---

Generally, TACPOL meets this requirement. Certain defaults are allowed however. See answer to Requirement E3 for some examples.

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

--- p ---

Generally, TACPOL meets this requirement. See I1 and E3 for exceptions.

Requirement: I3

The user will be able to associate compile-time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

--- F ---

TACPOL does not support these features.

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile-time variables. In such cases the condition will be evaluated at compile-time and only the selected path will be compiled.

--- F ---

Although TACPOL does have a SWITCH capability, it is not used in the manner described.

Requirement: I5

The source language will contain a simple, clearly identifiable base or kernel which uses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

--- F ---

TACPOL contains no such base or kernel.

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

--- P ---

The FORMAL DEFINITION of TACPOL explicitly denotes most of these restrictions, including the number of array dimensions and length of identifiers. Other translator restrictions (e.g., depth of nested parenthesis levels in expressions and maximum number of identifiers in a program) are not given in the FORMAL DEFINITION and cannot be explicitly defined b the user.

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

--- I ---

TACPOL meets these requirements.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run-time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

--- U ---

Run-time considerations (including object code efficiency) are not defined in the TACPOL FORMAL DEFINITION.

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

--- U ---

These considerations are undefined in the FORMAL DEFINITION.

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

--- p ---

TACPOL meets most of this requirement with its C/DE statement (which precedes a block of non-TACPOL statements).

TACPOL does not, however, support compile-time conditional statements of the sort described in Requirement I4.

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

--- p ---

Specify object presentation . . . . F
Specify time/space tradeoff . . . . T

TACPOL meets the requirement of time/space tradeoff with its ALIGN/PACKED option in declarations.

TACPOL does not support user defined object presentation of composite data structures.

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

--- p ---

TACPOL expands all procedures in-line, and does not explicitly support closed procedures. There is, however, a suggestion that pre-compiled external procedures can be called; this feature is not explicitly stated or described, however.

TACPOL Evaluation Summary

This section presents a summary of the findings of the evaluation of the TACPOL language. Strengths and weaknesses of the language are presented. Requirements for language modification necessary to meet DOD needs and functions are addressed.

TACPOL has a set of strengths which conform closely to the requirements of "TINMAN". Included in these are the following:

Well defined syntax -- The syntax of the TACPOL language cannot be altered by the user. The language is based on a relatively small set of fairly meaningful keywords, and these may not be abbreviated. The language is almost completely free format.

Strongly typed -- All identifiers used in a TACPOL program must be declared by type before they can be used. Type checking between parameters and arguments in procedures is enforced.

Block structured -- TACPOL supports user defined blocks, within which local scoping of identifiers is permitted.

Declaration and designation of variables, literals, and constants -- TACPOL meets virtually all of these requirements.

TACPOL also suffers from numerous significant weaknesses, primarily omission of certain required "TINMAN" features. These include:

Pointers are not supported.

User cannot define new types of operators or ennumeration sets.

There is no facility for describing the object environment or providing an external interface to the run-time machine.

Shortcomings exist in the definition of iterative and conditional control structures, including condition evaluation and a non-mandatory ELSE.

Recursion is not well defined.

Parallel processing is not supported, nor do any time dependent features exist.

Exception handling is inadequate.

The symbol = is used as an assignment as well as a relational operator.

There is a considerable amount of defaulting permitted,

not all of which is consistently applied.

The only TACPOL features which appear to exceed "TINMAN" specifications are:

The intrinsic procedure BOOL permits evaluation of two bit strings according to a "truth table".

The numbers and sophistication possible for data structures are considerable.

The syntax of TACPOL does appear to conform to the rules specified for the evaluation, i.e., little of the language syntax would have to be modified to eliminate clashes with "TINMAN" requirements.

TACPOL utilizes a fairly common set of constructs (in most cases). The language itself is free of machine dependent operations, and TACPOL programs could potentially be moved to a variety of machines. There are, however, a considerable set of assumptions about the compilation and target machines which may make such transportability difficult. (These include byte and word sizes, numeric precision, and an ASCII character set orientation.)

The conventions used for declaration and designation of complex data structures (e.g., cells) are difficult, not always perfectly consistent, and non-standard. It appears to be fairly easy to make a programming mistake in this regard.

TACPOL could be modified to meet most of the "TINMAN" requirements, but this would mean adding considerable syntax. Most such modifications would be straightforward, such as expanding numeric types, permitting division with remainder, adding "NOR", supporting address pointers, and changing the symbol used for relational equality. Others could be made with fairly easy language modifications, but may create serious translator and run-time difficulties, such as description of object environment, changing the iterative and conditional structure syntax rule (including making ELSE mandatory), defining recursive procedure calls, adding the syntax to support parallel processing, and tightening up currently permitted defaults. Finally, there are some changes which may require considerable new syntax and may also create significant translator and run-time costs, such as permitting user defined types, operators, and power sets of ennumeration, expanding exception handling, and providing I/O device definitions.

EVALUATION OF

CMS-2

Prepared by

RLG Associates, Inc.
11250 Roger Bacon Drive
Reston, Virginia 22090

November 18, 1976

This report presents an evaluation of the CMS-2 language with the requirements listed in the "TINMAN" (DOD Requirements for High Order Computer Programming Languages, "TINMAN" - 1 March, 1976, Section IV). For purposes of this comparison CMS-2 is considered to be defined by:

USER'S REFERENCE MANUAL (U) FOR COMPILER MONITER SYSTEM (CMS-2) FOR USE WITH AN/UYK-7 COMPUTER -- M-5035

CMS-2Y USER'S REFERENCE MANUAL (PRELIMINARY COPY) -- M-5049

Throughout this report, the use of the term CMS-2 denotes CMS-2Y, the version of the CMS-2 system designed for use with the AN/UYK-7 Computer.

There are 78 language requirements listed in Section 4 of the "TINMAN". This report compares CMS-2Y with each individual requirement. A summary of the degree to which the language satisfies each requirement is presented.

The introductory paragraph of each "TINMAN" requirement is included as the leading section for each requirement evaluation.

Symbols placed beside each individual requirement indicate the degree to which the language meets the requirement. The symbols and their meaning are as follows:

T - Totally meets requirement

P - Partially meets requirement

F - Does not meet requirement

U - Unknown from available documentation

A. DATA AND TYPES

Requirement: A1

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source program.

---p---

In general, CMS-2 is a typed language. Most types are known at compile time. However, there are a few instances where typing is not required by CMS-2. These are as follows:

Data structures can be defined to consist of machine-dependent word groups. In these cases, it is strictly up to the user to be cognizant of any data manipulation problems which may arise.

Additionally, the OVERLAY construct provides a method for violation of type checking mechanisms.

In CMS-2 formal parameters need not be explicitly typed. CMS-2 assumes a default numeric type for parameters.

Implicit variable declarations can be made in CMS-2. Again, CMS-2 assumes a default integer type for such declarations.

Type checking is performed at compile time. However, numeric data units are considered to be of the same type during assignment and evaluation of expressions; implicit conversions take place when two incompatible numeric units are manipulated. This fact coupled with the use of the OVERLAY construct provides the means for violation of type checking mechanisms. In this sense, CMS-2 does not meet the full intent of this "TINMAN" requirement.

Requirement: A2

The language will provide data types for integer, real

(floating point and fixed point), boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

---P---

Integer.....T
Fixed point.....T
Floating point.....T
Character string.....T
Boolean.....T
Arrays.....P
Records.....P

Arrays and Records -- CMS-2 permits both arrays (called TABLEs) and Records (also called TABLEs). CMS-2 does not however, provide the capability of using arrays and records as type generators. Thus in this sense, CMS-2 does not meet the full intent of the "TINMAN".

Requirement: A3

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

---F---

There is no language element to specify the precision to be used in floating point arithmetic operations. The default precision used in all floating point operations is that of the AN/UYK-7 computer.

The cost of implementing floating point precision specification should not be too great. However, programs currently core restricted may be affected adversely.

Requirement: A4

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor

management will be done by the compiler.

---T---

CMS-2 totally meets this requirement.

Requirement: A5

Character sets will be treated as any other enumeration type.

---P---

CMS-2 has a built in character type, which represents the ASCII set in an internal, implementation dependent, collating order.

The addition of EBCDIC and other widely used character sets, and the run time conversion packages to translate one set to another, could be done at modest cost to the existing implementations.

Requirement: A6

The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

---P---

Number of dimensions fixed at compile time.....T
Type fixed at compile time.....P
Lower subscript bound fixed at compile time....T
Subscripts from contiguous range.....T
Subscripts from enumeration type.....P
Upper subscript bound fixed at scope entry.....T

Since CMS-2 allows untyped word groups to be components of arrays, the language partially fails the requirement that the type of each array component be explicitly specified by the user.

Subscripts can only be integers in CMS-2. Since character sets are not an enumeration type (see A5 above) in CMS-2,

the language partially fails the requirement that subscripts can be from an enumerated type.

Requirement: A7

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

---T---

While records (TABLEs) can have alternative structures in CMS-2, there is no explicit requirement that all components of records be named, nor that they be typed. CMS-2 allows array components to be a group of contiguous machine words.

B. OPERATIONS

Requirement: B1

Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

---p---

Variable declaration for all data types.....T
Encapsulated type declaration.....F
Array or record declaration.....T

Encapsulated type definitons are not a part of CMS-2. All items in an array or table can be assigned using one statement in CMS-2.

Requirement: B2

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

---p---

Arrays cannot be compared in their entirety in CMS-2. CMS-2 assumes some type conversions when two distinctly typed objects are compared; for example, when a fixed-point data object is compared to a floating point object, the comparison would be done algebraically in floating point.

Requirement: B3

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

---p---

All six relational operators are included. Unordered sets

are not included in CMS-2.

Requirement: B4

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

---T---

CMS-2 meets this requirement.

Requirement: B5

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

---P---

There are no explicit range specifications within the CMS-2 language. The ranges are implicit in that the various types represent the data internally in 16,32 or 64 bit form.

There is only one precision for floating point numbers, and CMS-2 uses the AN/UYK-7 rounding instructions, when the rounding option is invoked.

Requirement: B6

The built-in boolean operators will include "AND", "OR", "NOT", and "OR". The operations "AND" and "OR" will be evaluated in short circuit mode.

---P---

Short circuit "AND".....T
Short circuit "OR".....T
NOT.....T

NOR.....F

The operation "OR" is not included within the language. Short circuit mode evaluation for scalars is defined within the language.

Requirement: B7

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

---P---

Assignment between records and arrays.....T
Scalar operations on arrays.....P

Data transfer between arrays and records are permitted in CMS-2. CMS-2 makes no distinction between records and arrays.

Scalar operations are permitted only on variables, and on individual elements of arrays and records.

Requirement: B8

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

---P---

Explicit conversions.....F
No implicit conversions.....F
Explicit between scale factors.....T

Implicit type conversions are used within the CMS-2 language when certain options are invoked; for example, in relational expressions, when an integer and a real number are compared, the integer would be converted to

the real format.
Explicit conversions are not permitted in CMS-2. Since all numeric data units are considered to be of the same type, internal conversions take the place of explicit conversions.

The format capability on output allows object representations of numbers to be decoded as characters.

Requirement: B9

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

---F---

The capability for run time exception checking can be included within the language.

Requirement: B10

The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

---F---

CMS-2 is oriented toward a serial batch processing environment. Its input/output statements are insufficient to handle real time applications.

No capability exists in CMS-2 for dynamic assignment/reassignment of I/O devices.

The addition of powerful I/O capabilities to CMS-2 would be a major effort.

Requirement: B11

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

---F---

Power sets are not defined in the language. This capability can be added without much difficulty.

C. EXPRESSIONS AND PARAMETERS

Requirement: C1

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

---T---

Arguments of an expression causing side effects are evaluated left-to-right.

Requirement: C2

Which parts of an expression constitute the operands to each operation, within that expression, should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

---P---

Unambiguous presentation.....P
Few precedence levels.....T
Require explicit parentheses.....F
No user defined levels.....T

CMS-2 has six precedence levels. There are some cases where the use of the unary minus operator leads to ambiguity. For example, -3**2 evaluates in CMS-2 to 9; but -A**2 where A has the value 3, evaluates to -9. This evaluation of exponentiation is done from right-to-left in the latter case, and is ambiguous at best.

Again, the rules for evaluation of substitution declarations are quite complex and confusing.

In most cases however, the operands to each operation are easily discernable.

Requirement: C3

Expressions of a given type will be permitted anywhere in source programs where both constants and references to

variables of that type are allowed.

---T---

There are no special case constraints on expressions included in the syntax of the language, except in the case of substitution declarations where the form is different.

Requirement: C4

Constant expressions will be allowed in programs where constants are allowed, and constant expressions will be evaluated before run time.

---T---

This requirement is fully met by the language.

Requirement: C5

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handling, for parallel processes, for declaration, or for built-in operators. There will be no special operations (e.g., array structuring) applicable only to parameters.

---P---

Consistency in parameter rules.....T
No special parameter operations.....P

CMS-2 has a consistent set of rules for all parameters. However, for the sake of efficiency, the language designers have included a clause allowing the use of the machine registers to pass parameters.

This language clause, "PARAMETER DECLARATION" associates one of the user specified AN/UYK-7 machine registers with a formal parameter. This clause is highly machine-dependent and is not in full agreement with the "TINMAN" requirement that there be no special operations applicable only to parameters.

Requirement: C6

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

---P---

Dimensions determined at compile time.....T
Size and subscripts can be passed.....P
Type agreement for actual and formal parameters.....P

In general there is agreement of type between formal and actual parameters. However, as pointed out in A1, all numeric data is considered to be of the same type.

There is no direct mechanism for passing of size and subscripts of arrays. However, through the use of the major index (size indicator) of simple tables (one dimensional arrays), the size of an array can be passed as a parameter. This is really a restrictive and hidden feature, and practically speaking, the language fails the intent of the "TINMAN" requirement.

Requirement: C7

There will be four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

---P---

Parameter constants.....P
Parameter variables.....P
Exception conditions.....T
Procedure parameters.....F

CMS-2 calls are made by value, except in the case of indirect tables, in which case the CORAD function call may be used to pass the address of a load-time allocated table. Actual parameter values are not affected by the call.

Exception conditions can be stated by the use of the "EX-

IT" clause. The "EXIT" clause allows for program control to be transferred to one of the named statement labels, which have to be within the scope of the calling procedure block.

Requirement: C8

Specification of the type, range, precision, dimension, scale and format of parameters will be optional on the formal side. None of them will be alterable at run time.

---F---

CMS-2 does not allow for deviation from the specified language rules for writing procedures.

It is not apparent to the reviewer how this requirement can be achieved and the requirements of A1 and A2 simultaneously achieved. Perhaps a special construct, GENERIC, to be used only by an elite tightly controlled group of programmers.

Support of this capability requires a significant change to the the CMS-2 language.

Requirement: C9

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

---P---

Variable number of arguments.....P
All but constant number same type.....F
Number fixed at compile time.....T

The number of arguments for all procedures is fixed at compile time. CMS-2 does allow for the omission of unrequired parameters on any given call to a procedure. This is accomplished by the use of commas to denote missing arguments in a call. However, it is up to the called pro-

cedure to be cognizant of the fact that a few parameters are missing. In this sense, one could state that CMS-2 allows for a variable number of arguments.

The requirement that all but a constant set of the variable number of arguments be of the same type is not enforced in CMS-2.

D. VARIABLES, LITERALS AND CONSTANTS

Requirement: D1

The user will have the ability to associate constant values of any type with identifiers.

---T---

The language provides this capability.

Requirement: D2

The language will provide a syntax and a consistent interpretation for literals of built-in data types. Numeric literals will have the same value (within the specified precision) in both programs and data (input or output).

---P---

Literals of built-in data types.....T
Consistency in value.....U

The existing documentation is unclear as to the consistency of values in both programs and data.

Requirement: D3

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

---P---

Declare initial values.....P
Allocation scope initialization.....T
No default values.....T

Variables and components of tables can be set to predefined values. However, whole arrays or tables cannot be preset. Local variables (local indexes in CMS-2) cannot be preset either.

There are no default values in CMS-2.

Requirement: D4

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

---p---

Numeric variable range specification mandatory.....F
Step size for fixed point numbers must be specified.....T
Range and step size not defining a new type.....T

No explicit range specifications are mandatory in CMS-2. However, the absolute maximum value a fixed point variable can take is implied by the number of bits requested in the definition. The minimum values cannot be specified.

The range for floating point numbers are not specifiable and default to the range allowed by the AN/UYK-7 computer.

Requirement: D5

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

---p---

Enumeration types.....F
Defined Types.....F
Built-in Types.....T

Variables, array components and Record components can assume values of the built-in data types.

No user defined types are allowed in CMS-2.

Requirement: D6
The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

---F---

The only pointer mechanism as such, available in CMS-2 is the CORAD function call which returns the address of the requested variable, table or array component. This could be used for reference and assignment purposes. The CORAD function is intended primarily for indirect tables.

The addition of a powerful pointer capability to CMS-2 will require significant modifications to the present implementations.

E. DEFINITION FACILITIES

Requirement: E1

The user of the language will be able to define new data types and operations within his programs.

---F---

CMS-2 is restricted to the use of built-in data types. The definition of new types and operations as intended here are not supported by the language.

Requirement: E2

The "use" of defined types will be indistinguishable from built-in types.

---F---

The language does not permit definition of new data types as an operation.

This capability can be added to the language as an extension of E1.

Requirement: E3

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

---P---

Local variables can be declared implicitly in CMS-2. They default to a signed-integer type.

Removal of such implicit declarations should be trivial.

Requirement: E4

The user will be able, within the source language, to extend existing operators to new data types.

---F---

The language does not permit new type definition. This capability can be included as part of the extension to type definition.

Requirement: E5

Type definitions in the source language will permit definition of both the class of data objects comprising the types and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

---F---

The language does not support user defined types.

Requirement: E6

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

---P---

Enumeration.....F
Cartesian products.....T
Discriminated union.....F
Power set of enumeration type....F

The TABLE mechanism is used to define records. Its use is limited, however. Records do not define types and cannot be used as type constructors.

Requirement: E7

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

---F---

Does not permit free unions.....F
Does not permit subsetting.....F

Free unions are permitted with the use of the OVERLAY capabilities in JMS-2.

Subsetting is possible by the use of the SUB-TABLE mechanism.

Requirement: E8

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

---F---

New types cannot be defined within the language.

F. SCOPES AND LIBRARIES

Requirement: F1

The language will allow the user to distinguish between scope of allocation and scope of access.

---F---

In CMS-2, the scope of allocation of all data structures is the duration of the program, with the exception of local variables (local indexes) where the variable is accessible for the duration of the defining procedure.

The scope of access for all data structures is, again, throughout the program for data declared in system-data-designs (global definitions); for data declared in local-data-designs the scope of access is limited to those groups of procedures; for local indexes, the scope of access is limited to the procedure defining the local index.

The modifications required to allow a true local/global definition facility would be significant, since the required changes would have to allow for both allocation and deallocation of data structures.

Requirement: F2

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

---F---

As stated above, access to data structures is limited by the physical location of the data declarations. The only exception to this rule is as follows:

By using the EXTDEF and EXTREF clauses data structures declared locally can be made globally accessible.

The assessment of the cost and impact of the modifications necessary to incorporate this requirement can be made in conjunction with the modifications neccessary for

implementing requirement F1.

Requirement: F3

The scope of identifiers will be wholly determined at compile time.

---T---

This requirement is completely met by the language.

Requirement: F4

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

---T---

Applications oriented data and subroutines can be stored in common libraries which are easily accessible by CMS-2.

Requirement: F5

Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

---P---

Accessible at compile time.....T
Other source language routines.....P

The language does not preclude routines written in a different source language. As long as the routines exist in a source form compatible with the CMS-2 library manipulators they can reside in system libraries. There is no provision for interface checking in the language. Hence, it cannot be said whether foreign language routines can exist in object form in the system libraries.

Requirement: F6

Libraries and compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

---P---

Libraries and compools indistinguishable.....P
Hold anything definable in language.....T
Many levels of access.....T
Many specialized subsets.....T

CMS-2 libraries and compools can hold anything definable in the language. However, for elements being selected from a compool, the user has to state that a compool is being selected.

Requirement: F7

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

---F---

CMS-2 has a limited high level I/O capability. A limited set of peripheral equipment and special hardware is supported. See comments under B10.

G. CONTROL STRUCTURES

Requirement: G1

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

---P---

Sequential control.....T
Conditional control.....T
Iteration.....T
Recursion.....F
Parallel processing.....F
Exception handling.....P
Asynchronous interrupts.....F

The language provides the following keywords for control mechanisms:
FOR
GOTO
IF . . . . THEN . . . . ELSE
VARY

Exception handling is provided by use of the abnormal exit and the ability to alter the sequence of execution overflow upon condition detection of parameters.

The language does not provide an explicit mechanism for interrupt handling.

Similarly, there is no explicit definition of parallel processing. Addition of this to the structure of the language will require major language modification.

In general, the language control mechanisms conform to "TINMAN" requirements. The addition of the other "TINMAN" capabilities to CMS-2 would be a non-trivial effort.

Requirement: G2

The source language will provide a "GOTO" operation ap-

plicable to program labels within its most local scope of definition.

---p--

CMS-2 permits the GOTO target to be any label in the system-procedure.

Rules governing the GOTO operation can be modified to preclude branches out of scope.

Requirement: G3

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

---p---

Based on Boolean expression.....T
Based on type from discriminated union.....F
Based on computed choice.....T

Conditional control based on Boolean expressions is provided for by the *IF-THEN-ELSE construct*. Case statements and switches provide for a computed choice among labeled alternatives.

The language does not require that all alternatives be accounted for; for example, an "ELSE" clause is not mandatory for all "IF-THEN" expressions.

There are no simple mechanisms for common cases.

Requirement: G4

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity, or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

---p---

Termination anywhere in loop.....T
Local control variables.....F
Entry at loop head only.....F
Efficient and clear simple cases.....T
Control value available at termination.....P

Termination within a loop is made possible by several constructs within the language. Termination can be at the head of the loop (after a fixed number of iterations or failure of a test - WHILE), at the end of the loop (test - UNTIL) and within the loop (using the GOTO).

Loops can be entered at any labeled statement.

The control variable may or may not occur within the controlled statement. The controlled variable is a word reference, i.e., either an anonymous reference or a declared word reference. The value of the controlled variable is available for declared word references.

The rules for iterative evaluation can be redone to completely meet "TINMAN" requirements.

Requirement: G5

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

---F---

Recursive and nonrecursive routines.....F
Explicitly specify recursive routines.....F
No nested recursive procedures.....F

Recursion is not provided in CMS-2.

Requirement: G6

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possible pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

---F---

The language does not explicitly address parallel processing requirements.

Requirement: G7

The exception handling control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

---F---

CMS-2 does not provide a general exception handling capability. The only constructs available are abnormal exit parameters and overflow conditions.

Addition of exception control constructs to the language will be a nontrivial process.

Requirement: G8

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, and which permit asynchronous hardware interrupts to be treated as any other exception situation.

---F---

This requirement appears to expand on the G6 requirements. Parallel processing capabilities can be added to the language. This addition will be costly and may tend to cloud the readability and clarity of the language.

H. SYNTAX AND COMMENT CONVENTIONS

Requirement: H1

The source language will be free format with an explicit statement separator, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple, uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

---P---

Free format with statement separator.....T
Mnemonically significant identifiers.....T
Conventional forms.....P
Simple uniform grammar.....P
No special case notations.....P
No abbreviation of keywords or identifiers....T
Unambiguous grammar.....P

CMS-2 is a free-format language. There are a number of special cases used in the grammar of the language, for example, the rule for Boolean operations is quite complex. The rules for substitution expressions are different from normal usage.

Requirement: H2

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new infix operator precedence.

---T---

The syntax is completely fixed.

Requirement: H3

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using

the 64 character ASCII subset.

---T---

The language satisfies this requirement.

Requirement: H4

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

---P---

Literals for numbers.....T
Character strings.....T
Break character.....F

Identifiers can be from one to eight characters in length. However, the first character has to be an alphabetic.

There are no break characters, per se, in CMS-2. As a consequence, there are elaborate rules as to where a space can be used, and the usage is context dependent. Break characters could be introduced into the language with minor modifications.

Requirement: H5

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

---F---

Multiple lines are permitted within the language structure. It should be very simple to change this feature.

Requirement: H6

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

---P---

The language defines 179 key words. The proliferation of key words is highly indicative of the complexity of the language.

Requirement: H7

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

---P---

Single comment convention.....F
Distinguishable from code.....P
Introduced by one or two characters.....P
Contain any character.....T
Appear anywhere reasonable.....T
Terminate at end of line.....F
Not prohibit reformatting.....T

Comments may be expressed in three forms:

COMMENT <string>

Bracket Notes: These are specialized documentation features associated with system declarations and data declarations.

Strings quoted by apostrophes: These can appear anywhere a space can within a statement.

These three forms can be replaced with just one of these forms, with minor effort.

Requirement: H8

The language will not permit unmatched parentheses of any kind.

---T---

The language does not permit unmatched parentheses.

Requirement: H9

There will be a uniform referent notation.

---P---

All notation is consistent, except for the intrinsic functions, BIT and CHAR.

Requirement: H10

No language defined symbols appearing in the same context will have essentially different meanings.

---P---

There are several cases where the meaning of language symbols is context dependent.

- AND and OR are both logical connectors and bit-string operators.

- THEN is both a statement connector and a conditional statement connector, i.e., IF ... THEN.

- EQUALS has three meanings - numeric constant EQUALS, the difference in addresses between allocated identifiers and the allocation of an identifier.

This leads to considerable ambiguity in usage. Changes to eliminate this confusion would require changing both the syntax as well as the translators.

I. DEFAULTS, CONDITIONAL COMPILATION, AND LANGUAGE RESTRICTIONS

Requirement: I1

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

---T---

This requirement is satisfied by the language.

Requirement: I2

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

---P---

In general the language does not permit defaults. However, there are some exceptions. For example, the mode of packing of TABLE fields can be selected by the programmer or he can let the compiler manage it optimally. Reentrant procedures have to be explicitly specified or the mode is non-reentrant.

Requirement: I3

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

---F---

CMS-2 does not allow this capability. The only available compile time feature is the ability to selectively compile pieces of source code. Addition of this capability would require major modifications.

Requirement: I4

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the condition will be evaluated at compile time and only the selected path will be compiled.

---F---

Selective compilation is possible using the CSWITCH mechanism. This allows for compilation only when the CSWITCH "flag variable" is set on. This capability does not meet the full intent of the "TINMAN" requirement.

Requirement: I5

The source language will contain a simple, clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capability not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

---F---

As far as can be determined, the base or kernel constitutes the entire language.

Requirement: I6

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

---P---

Available documentation describing compiler limits is precise in some areas and vague in others. For example, array dimensions, identifier lengths, and nesting limits are explicitly defined. The number of identifiers allowable is not explicitly defined.

Limits are probably dependent upon the machine environment in at least some cases.

Requirement: I7

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

---T---

The language satisfies this requirement.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

Requirement: J1

The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

---U---

It is not clear from the documentation as to how the code generators function. Also, the evaluator cannot determine whether run-time support routines are used.

Requirement: J2

Any optimizations performed by the translator will not change the effect of the program.

---T---

Insofar as it is known to the evaluator, optimization done by the CMS-2 compiler does not change the effect of the program.

Requirement: J3

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

---T---

The DIRECT CODE construct provides this capability.

Requirement: J4

It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation

will be optimal as determined by the translator.

---P---

Specify object presentation.....P
Specify time/space tradeoff.....P

This capability is somewhat available through use of the TABLE construct. Field orders, field widths, bit patterns and field structures can all be specified within a table structure.

There is no mechanism for explicitly specifying tradeoffs. However, the translators do in fact optimize data manipulations for the object computer.

Requirement: J5

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and closed routine of the same description will have identical semantics.

---F---

All user defined procedures and functions are closed. The language could be modified to add this capability.

CMS-2 Evaluation Summary

This section presents a summary of the findings resulting from this evaluation of CMS-2. The strengths and weaknesses of the language are noted. The feasibility of modifying the language to fully meet the "TINMAN" requirements is discussed.

CMS-2 has the following strong points in comparison with the "TINMAN".

- Well Defined Syntax -- syntax of the language cannot be changed; no keyword abbreviations; language is free format.

- Binding Capability -- programs may be compiled separately and linked at execution time; COMPOOLS can be wed to hold intermediate output.

The major weaknesses of the language in comparison to the "TINMAN" are as follows:

- Parallel processing not supported.

- A general exceptional handling capability is not available in CMS-2.

- Powerful I/O capabilities, as required by the "TINMAN", are not supported.

- No facilities for user definable data types, and the associated ability to define new operations for such data types.

- Strong typing, as required by the "TINMAN", is not rigidly enforced in CMS-2.

In general, CMS-2 provides an adequate framework for

problem solving. Since the base, or kernel, of the language constitutes the entire language, it is a fairly complex one. For example, the documentation lists 179 key-words. The language allows some machine-dependent operations; such operations include register use, word size and internal data representations. Certainly these constructs can be removed, but at some expense. As is the case with other languages implemented on a variety of machines, a number of implementation dependencies have crept into CMS-2. Applications programs taking advantage of such installation dependent features may have to be modified to operate on other machines.

CMS-2 programs tend to be hard to read, due to the use of a limited character set (026 codes), as well as the use of three distinct comment conventions. Since maintainability is a major DOD goal, readability of programs is highly desirable.

Modification of CMS-2 to fully meet the DOD "TINMAN" requirements is possible. Some features, such as an extended pointer capability and automatic range checking, can be easily included. Other features, such as expanded array and record handling capabilities, recursive control structures, powerful I/O capabilities, parallel processing and strong typing could prove to be costly, both in terms of time and money. In fact, the modifications necessary to incorporate the latter capabilities would destroy much of the flavor of CMS-2, as it is known today.